МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра Прикладная математика и информатика
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки / специальности)

Управление корпоративными информационными процессами
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему Архитектурные решения при построении легковесных корпоративных веб-приложений

Обучающийся В.В. Авчинников
(Инициалы Фамилия) (личная подпись)

Научный руководитель доцент, канд. пед. наук, О.М. Гущина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Оглавление

Введение

Сегодня сфера информационных технологий развивается довольно стремительно. Вместе с ней развивается и сфера интернет-сервисов. С каждым днем появляется все больше и больше веб-сайтов и веб-приложений для простых пользователей. Не отстает и корпоративная сфера, переводя внутренние информационные процессы в сферу веб-технологий и облачных решений. Вместе с этим растет и сложность корпоративных веб-приложений, наполненных разнообразным функционалом, среди которого чаты, таблицы, формы, диаграммы и множество другого. Но это богатство функционала, естественно, имеет немалый размер и требует для комфортного использования устройства с хорошими техническими характеристиками и высокоскоростным интернетом. Если же у пользователя интернет не отличается высокой скоростью, то такое приложение будет долго загружаться, а на бюджетных устройствах с низкой производительностью могут наблюдаться подтормаживания или вообще зависания из-за нехватки доступной оперативной памяти

Актуальность исследования обусловлена потребностью в обеспечении комфортного использования веб-приложений, включая корпоративные, на любых устройствах, независимо от их производительности и наличия высокоскоростного интернета.

Согласно статистически данным, в декабре 2024 средняя скорость мобильного интернета составила в целом по стране 25 Мбит/сек [32]. Тут нужно отметить, что средняя скорость разная в зависимости от региона. Так удалось найти данные на конец 2023 года, в которых указано, что в Москве средняя скорость была 63 Мбита/сек, а по России без учета Москвы всего лишь 18,7 Мбит/сек [7]. Причем, нужно понимать, что 18,7 – это средняя скорость вне Москвы, у одного пользователя может быть скорость интернет-соединения 30 Мбит/сек, а у другого – 6, в среднем получится – 18.

В интернете можно найти множество негативных комментариев/отзывов людей о мобильном интернете от разных операторов, поэтому мы понимаем, что в отдельных довольно разнообразных случаях есть проблемы.

Объектом исследования является корпоративное веб-приложение.

Предметом исследования – методы сокращения размера клиентской части корпоративных веб-приложений.

Гипотеза исследования – перенос логики пользовательского интерфейса из клиентской в серверную часть корпоративного веб-приложения позволит сократить размер веб-приложения на устройствах пользователей, что приведет к ускорению загрузки и улучшению производительности работы приложения в целом.

Целью работы является исследование существующих и выработка новых архитектурных подходов для создания легких веб-приложений.

Задачи, которые будут выполнены в ходе проведения исследовательской работы:

- анализ истории и современного состояния предметной области;
- обзор научных работ и других публикаций по теме архитектуры и способов сокращения размера клиентской части веб-приложений;
- анализ эффективности известных способов оптимизации размера клиентской части веб-приложений;
- выработка альтернативной архитектурной модели, позволяющей сократить размер клиентской части веб-приложения, ее апробация и оценка эффективности.

В ходе научно-исследовательской работы применялись следующие методы: метод анализа, измерения и сравнения, табличный и графический методы.

Научно-исследовательская работа проводилась в период с 2023 по 2025 год и состояла из нескольких этапов.

Первый этап заключался в исследование предметной области. На этом этапе было проведено исследование истории развития и современного

состояния сферы веб-приложений, была определена тема исследования, гипотеза, цели и задачи, предмет и объект исследования.

На втором этапе был произведен поиск и анализ научных работ и других публикаций по теме архитектуры веб-приложений и методов оптимизации скорости их загрузки и скорости их работы для определения степени изученности проблемы и имеющихся решений.

На третьем этапе были определены методы исследования, подходящие для поиска и выработки нового решения поставленной проблемы, была выработана архитектурная модель, направленная на сокращение размера клиентской части веб-приложений за счет размещения всей логики пользовательского интерфейса в серверной части веб-приложения с последующей ее передачей на клиентскую часть в формате JSON при необходимости отображения конкретной части веб-интерфейса.

На четвертом этапе была произведена апробация гипотезы исследования и оценка эффективности ее результатов. Для этого было разработано демонстрационное веб-приложение, построенное согласно выработанной модели и произведено сравнение с тремя другими веб-приложениями разных размеров, построенными без использования предложенной модели.

По результатам отдельных этапов научно-исследовательской работы была опубликована статья «Проблемы размещения бизнес-логики в клиентской части веб-приложений» в журнале «Молодой ученый» [1] и выпущен доклад «Сокращение бизнес-логики в клиентской части веб-приложений» на XC Международной научной конференции в городе Казань [2].

Научная новизна работы заключается в предложении новой архитектурной модели веб-приложения, направленной на исключение логики пользовательского интерфейса из клиентской части веб-приложения с целью ускорения его загрузки и снижения расхода оперативной памяти.

Теоретическая значимость научно-исследовательской работы заключается в том, что работа вносит вклад в развитие архитектуры веб-

приложений, предлагая новую модель размещения логики пользовательского интерфейса, расширяя перечень доступных методов сокращения размера клиентской части веб-приложений.

Практическая значимость данной работы заключается в разработке и апробации архитектурной модели, позволяющей минимизировать размер клиентской части веб-приложений за счёт переноса логики пользовательского интерфейса на серверную сторону. Это способствует снижению требований к ресурсам устройств пользователей, обеспечивая более стабильную работу веб-приложений на низкопроизводительных устройствах и в условиях медленного интернет-соединения.

На защиту выносятся:

– модель веб-приложения с размещением логики пользовательского интерфейса в серверной части;
– результат апробации модели веб-приложения с размещением логики пользовательского интерфейса в серверной части, включая оценку изменения скорости загрузки и расхода оперативной памяти на устройстве клиента.

Структура и объем работы: 78 страниц; 45 рисунков; 6 таблиц; 35 использованных источников.

Глава 1 Теоретические основы сферы веб-приложений

1.1 Основные понятия, история и развитие веб-приложений

Сейчас веб-ресурсы отличаются большим разнообразием, есть статические веб-сайты, есть веб-сайты с небольшим интерактивным функционалом, а есть и веб-приложения, такие как онлайн редактор таблиц, онлайн редактор макетов дизайна. Объем и сложность функционала некоторых веб-приложений сопоставим с обычными программами, устанавливаемыми на компьютер или другое вычислительное устройство, например планшет или мобильный телефон.

Но для хорошего понимания предметной области, необходимо сделать краткий экскурс в историю. В привычном для нашего понимания первый веб-сайт появился в далеком 1991 году. Конечно, это был не красочный сайт с красивым дизайном, а всего лишь набор страниц с текстом и ссылками.

«Первый в мире сайт info.cern.ch появился 6 августа 1991 года. Его создатель, Тим Бернерс-Ли, опубликовал на нём описание новой технологии World Wide Web, основанной на протоколе передачи данных HTTP, системе адресации URI и языке гипертекстовой разметки HTML. Также на сайте были описаны принципы установки и работы серверов и браузеров. Сайт стал и первым в мире интернет-каталогом, так как позже Тим Бернерс-Ли разместил на нём список ссылок на другие сайты» [11].

Говоря простым языком, веб-сайт – это набор веб-страниц. В свою очередь, веб-страница – это документ или информационный ресурс, открываемый в веб-браузере. Веб-страница состоит из html-разметки и может включать в себя текст, изображений, ссылки и другие типы содержимого. HTML – это язык гипертекстовой разметки документов. Для перехода от одной страницы к другой используются ссылки, при клике на которые браузер открывает новую веб-страницу.

Код разметки простейшей веб-страницы представлен на рисунке 1, конструкции разметки в треугольных скобках называются тегами.

```
<!doctype html>
<html lang="ru">
  <head>
    <meta charset="UTF-8">
    <title>Простая html страница</title>
  </head>
  <body>
    <div>
      Текст обычной html-страницы и ссылка на
      <a href="/other-page/">другую страницу</a>
    </div>
  </body>
</html>
```

Рисунок 1 – Html разметка простейшей веб-страницы

Веб-страницы первых веб-сайтов не могли иметь какую-то оригинальную стилизацию, веб-браузеры того времени на свое усмотрение стилизовали элементы на страницах, например, определяли какой размер шрифта будет у заголовков, какие отступы у абзацев, нумерованных списков и так далее. Чтобы предоставить возможность стилизации веб-страниц на усмотрение автора, в 1996 годы была разработана рекомендация по первой версии стандарта CSS.

«CSS (англ. Cascading Style Sheets «каскадные таблицы стилей») – формальный язык декодирования и описания внешнего вида документа (веб-страницы), написанного с использованием языка разметки (чаще всего HTML или XHTML).» [21]

В первой версии рекомендации были следующие возможности для стилизации: «Параметры шрифтов. Возможности по заданию гарнитуры и размера шрифта, а также его стиля — обычного, курсивного или полужирного.

Цвета. Спецификация позволяет определять цвета текста, фона, рамок и других элементов страницы. Атрибуты текста. Возможность задавать межсимвольный интервал, расстояние между словами и высоту строки (то есть межстрочные отступы). Выравнивание для текста, изображений, таблиц и других элементов. Свойства блоков, такие как высота, ширина, внутренние (padding) и внешние (margin) отступы и рамки.» [21].

В последующих спецификациях CSS добавились новые и расширялись имеющиеся возможности стилизации, например была добавлена возможность скруглять углы и устанавливать фоновое изображение для рамок, возможность использования новых, более удобных методов позиционирования элементов, а также множество других полезных возможностей, которые в настоящий момент обеспечивают воплощение самых смелых дизайнерских идей.

С целью взаимодействия с пользователем, то есть с человеком, просматривающим веб-страницу в браузере, в html есть такой элементы – как форма. Форма состоит из разнообразных полей, например, однострочное текстовое поле, многострочное текстовое поле, поле выбора одного значения (чекбокс), поле выбора одного из нескольких значений (радиокнопка), поле выбора файла для последующей загрузки и некоторые другие поля. После заполнения необходимых полей пользователь нажимает на кнопку отправки, после чего данные из формы отправляются с помощью http запроса по заранее установленному веб-адресу.

Простое взаимодействие с пользователем на веб-странице через имеющиеся html-элементы довольно ограничено в плане функциональности. Например, на чистом html не реализовать добавление/показ дополнительного поля, если пользователь поставил галочку в другом. Для более сложных вариантов реализации логики взаимодействия на html-страницах используется скрипты на языке программирования javascript.

HTML, CSS и javascript – это базовые технологии, которые применяются в сфере веб-разработки.

С развитием веб-сферы понемногу росла и сложность веб-сайтов. Первые сайты состояли из набора статических страниц, которые в готовом виде лежали на веб-сервере. Затем появилась необходимость в динамической генерации страниц. Одним из самых популярных языков программирования для этой цели стал интерпретируемый язык программирования PHP, основательно переработанный в 1998 году. В настоящее время самым часто используемым языком программирования для серверной части веб-сайтов является PHP, так как на нем написано множество готовых систем управления контентом под разные нужды. Но также для разработки серверной части веб-приложений используются и другие языки программирования, например, Python, Ruby, Java, ASP.NET и другие. [35]

По мере роста взаимодействия веб-сайтов и их пользователей появился термин веб-приложения. Четкой границы между веб-сайтом и веб-приложением провести нельзя, но можно выделить ряд ключевых особенностей, которые присущи веб-приложению. Веб-приложение в большинстве случаев требует обязательной аутентификации и позволяет пользователям манипулировать контентом или другими сущностями (создавать/редактировать/удалять) в то время, как веб-сайт зачастую представляет из себя просто массив контента, который доступен для просмотра всем желающим.

1.2 Архитектура веб-приложений

Веб-приложения в большинстве случае имеют в своей основе клиент-серверную архитектуру. Как правило, данные и правила доступа к ним располагаются на стороне сервере, а логика их отображения и взаимодействия с пользователем находится в клиентской части веб-приложения.

Клиентская часть веб-приложений состоит из html-файла с подключенными к нему файлами css, содержащими правилами оформления, и файлами скриптов javascript – содержащими код, отвечающий за логику

пользовательского интерфейса. Также помимо вышеуказанных файлов могут подключаться файлы картинок, видео, аудио и других необходимых ресурсов для работы веб-приложения. Упрощенная схема веб-приложения показана на рисунке 2.

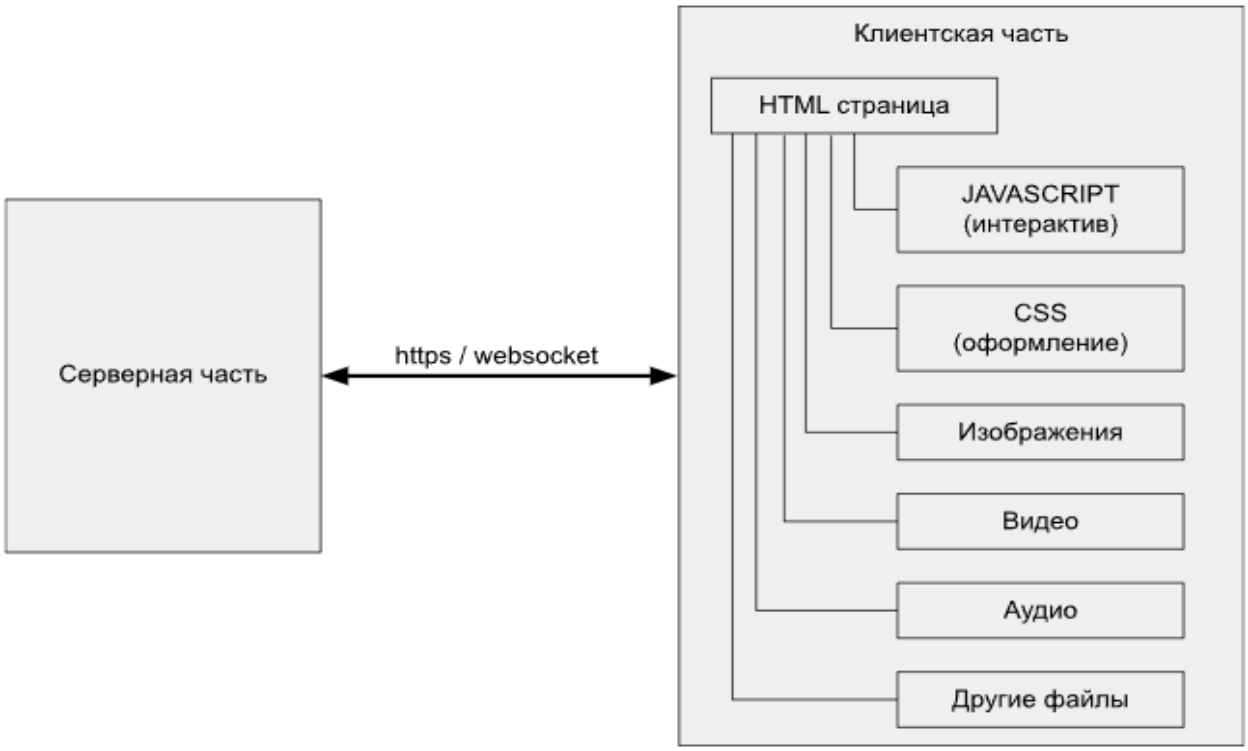

Рисунок 2 – Упрощенное представление веб-приложения.

Первое, что происходит при начале работы веб-приложения – это загрузка браузером с сервера html-страницы. После ее получения и анализа, браузер строит DOM (Document Object Model) дерево элементов, параллельно загружая дополнительные файлы, которые подключены на html-странице. Медленная скорость загрузки чего-либо (в том числе и веб-приложения) заставляет пользователя терять время в ожидании, что негативно сказывается в целом на опыте использования, поэтому необходимо стремиться к сокращению размера файлов, которые требуются для начальной загрузки веб-приложения.

Далее, по окончанию загрузки каждого ресурса, браузер выполняет необходимое для ресурса действие. После загрузки css-файлов стилей браузер строит CSSOM (CSS Object Model) и добавляет оформление элементам на html-странице. После загрузки изображений и других ресурсов контента, они появляются на странице.

Загруженные файлы javascript браузер сначала парсит, затем компилирует javascript-код в байт-код и сразу же выполняет его. Современные браузеры научились обрабатывать javascript файлы довольно быстро, но все равно сохраняется прямо пропорциональная зависимость между размером js-файла и временем на его обработку перед выполнением. Упрощенная модель взаимодействия серверной и клиентской части многостраничных веб-приложений представлена на рисунке 3.

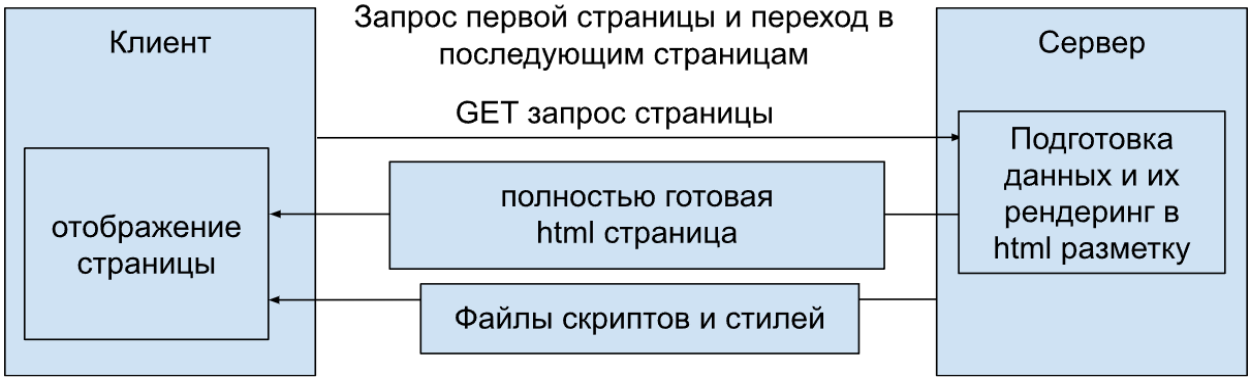

Рисунок 3 - Модель взаимодействия между серверной и клиентской частями многостраничных веб-приложений

Основным недостатком многостраничных веб-приложений является необходимость полной загрузки страниц при переключении страниц внутри веб-приложения, что занимает некоторое время и невозможно достижения плавности интерфейса, как в обычных приложениях. Именно поэтому вполне естественным шагом в процессе эволюции веб-приложений стало появление одностраничных веб-приложений, в которых переход к следующей странице происходит без полной загрузки страницы, а обновляется лишь контентная

часть страницы, при этом верхняя (header), нижняя (footer) и боковая (sidebar) не меняются. На рисунке 4 представлена модель взаимодействия в одностраничном веб-приложении.

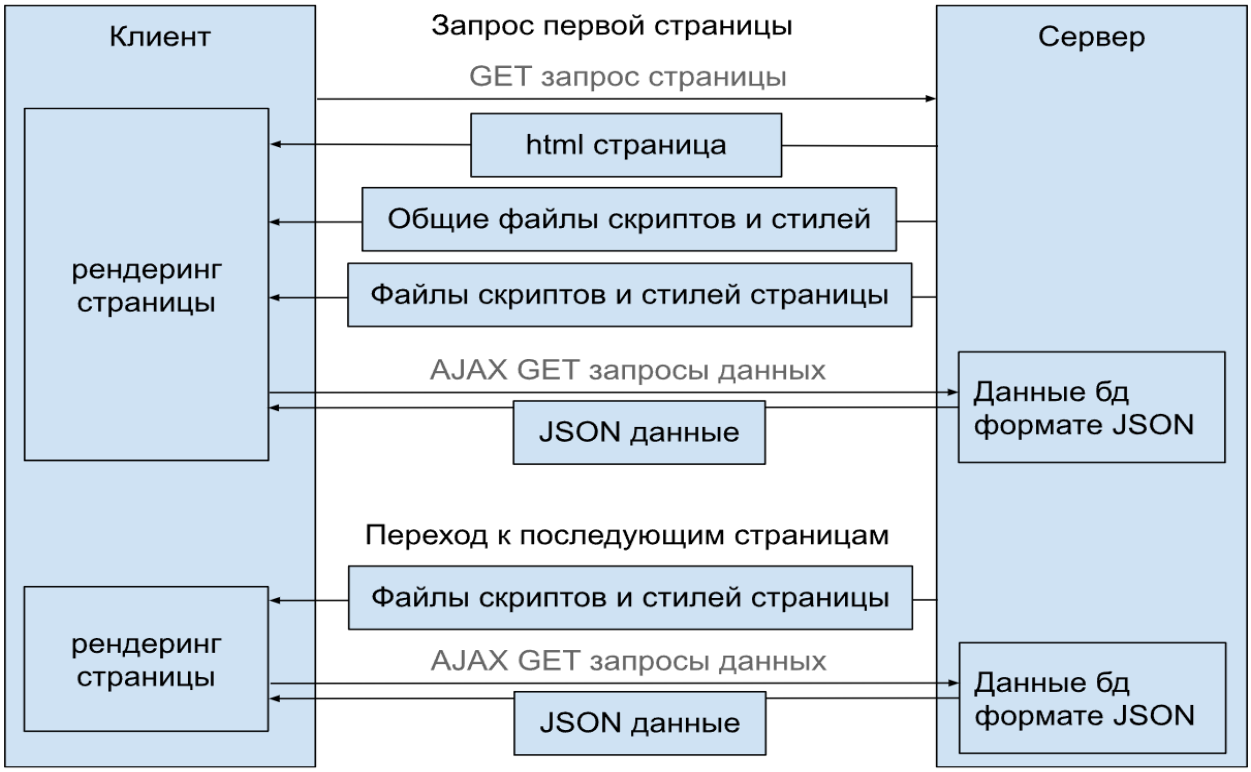

Рисунок 4 - Модель взаимодействия серверной и клиентской частей одностраничного веб-приложения

Корпоративные веб-приложения создаются для людей (сотрудников или клиентов компаний). Чтобы приложением было удобно пользоваться, его интерфейс должен быть интуитивно-понятным, оно должно работать стабильно и быстро. Скорость работы приложения зависит как от скорости ответа сервера, так и от быстродействия выполнения кода клиентской части приложения. В отличие от серверной части приложения, для которой необходимые ресурсные мощности могут быть легко увеличены, клиентская часть приложения работает на устройствах сотен, тысяч или даже миллионов пользователей. Устройства пользователей могут быть разными по производительности, объему оперативной и постоянной памяти, и чтобы у

всех пользователей веб-приложение работало хорошо, необходимо вести разработку с прицелом на использование низко-производительных устройств с низкой скоростью интернета.

Проблема большого объема кода клиентской части веб-приложений появилась вместе с широким распространением самих веб-приложений сравнительно недавно. Это обусловлено как в целом развитием веб технологий, так и появлением первых javascript фреймворков, таких как Backbone, Angular в 2010 году и Ember в 2011 году. Веб-приложения, как правило, и создаются на основе одного из javascript фреймворков, которые задают некоторые архитектурные принципы. В большинстве случаев веб-приложение строится с использованием паттерна проектирования MVVM.

MVVM – это сокращение от Model-View-ViewModel. Это паттерн архитектуры, который используется для построения пользовательского интерфейса в приложениях, его схема приведена на рисунке 5.

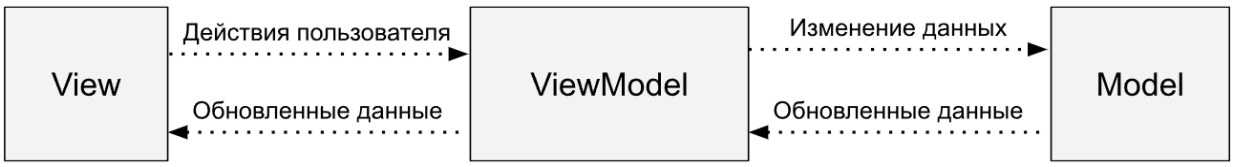

Рисунок 5 – Визуальное представление паттерна MVVM

Паттерн MVVM позволяет разделить код на три основных компонента:

– Модель (Model) представляет данные и бизнес-логику приложения. Он отвечает за получение, обновление и хранение данных, а также за их валидацию и обработку. В веб-приложении модель занимается синхронизацией данных между сервером и клиентской частью.
– Представление (View) отвечает за отображение данных и взаимодействие с пользователем. Это то, что пользователь видит на экране: элементы управления, макеты, графика и так далее.

– ViewModel – это посредник между моделью и представлением. Он обеспечивает связь между данными из модели и отображением в представлении. ViewModel предоставляет данные и команды, которые представление может использовать для отображения и взаимодействия с данными.

Веб-приложения в большинстве случаев представлены в виде одностраничного приложения (Single Page Application – сокращенно SPA). При таком подходе в начале загружается только одна HTML-страницу с сервера, а затем javascript код динамически обновляет содержимое без загрузки последующих страниц, как это было бы в случае с обычным веб сайтом. В SPA весь необходимый код HTML, CSS и JavaScript загружается однократно, а далее все взаимодействие с сервером и обновление пользовательского интерфейса осуществляются с помощью асинхронных (фоновых) запросов к серверу. С одной стороны – хорошо, что контент страницы обновляется без загрузки новой страницы, но с другой – плохо, так как в начале приходится ждать загрузки большого количества кода, который нужен не только для текущей страницы, но и для множества других. Такой подход как раз и породил проблему большого размера клиентской части веб-приложений.

При создании веб-приложений в настоящее время широко используются различные UI-фреймворки. UI – сокращение от user interface, что переводится как пользовательский интерфейс. Соответственно UI-фреймворки – фреймворк для разработки пользовательского интерфейса. Так как речь идет о веб-приложениях, то соответственно UI-фреймворк предназначен для разработки пользовательского интерфейса в браузере и реализуется на языке программирования javascript.

На данный момент в коммерческой разработке наиболее распространены фреймворки Vue, Angular и библиотека React, которая в связке с другими библиотеками, например react-router и redux предоставляет набор возможностей для разработки, сопоставимый с полноценным

фреймворком. Каждая из фреймворков имеет свои особенности, но в целом схожи тем, что реализуют реактивность, управление состоянием и удобный для переиспользования компонентный подход. Естественно, использование UI-фреймворка дает не только преимущества, но и некоторые недостатки, такие как увеличение размера кода клиентской части веб-приложения и увеличение потребляемых ресурсов на устройстве конечного пользователя. Стоит отметить, что есть такие фреймворки, которые имеют крайне малый размер и при работе приложения незначительно увеличивают использование ресурсов устройства, но в коммерческой разработке они используются крайне редко или вообще не используются.

1.3 Обзор публикаций по теме архитектуры веб-приложений

По теме архитектуры веб-приложений есть множество научных публикаций, в которых рассматриваются общие подходы построения веб-приложений. Например, в статье «Сравнение различных архитектурных решений при разработке веб-приложения» [3] авторы Вершинин Е. В. и Хромов А. Е. описывают два типа веб-приложений: одностраничное и многостраничное. Одностраничный тип веб-приложения является более современным, так как позволяет переключаться между разными страницами без перезагрузки веб-страницы в браузере, что гораздо ускоряет работу приложения в целом.

В статье «Особенности клиент-серверной архитектуры при реализации одностраничного web-приложения» Статников А. С. и Фролов Д. А. описывают два архитектурных подхода, используемых при реализации веб-приложений, относительного того, на какой стороне реализуется генерация html-кода, который отображается при старте веб-приложения.

«Серверный рендеринг подразумевает расположение всех основных слоев архитектуры, таких как логика обращения к базе данных, бизнес-логика и логика создания представления (интерфейса) на стороне сервера. Клиентская

сторона получает уже сгенерированное представление, показывает его пользователю, а также обрабатывает входные команды и отправляет их на сервер, который в качестве результата будет каждый раз возвращать новое представление. Как можно заметить, основным достоинствами является минимальная нагрузка на клиентскую часть (устройства пользователей), а также полная изоляция бизнес-логики от клиента, что позволяет обеспечить высокий уровень целостности и безопасности данных.

К недостаткам же можно отнести высокую нагрузку на сервер из-за того, что он выполняет большую часть работы по обработки данных приложения. Также сервер формирует представление множеству пользователей одновременно. Это недопустимо в условиях большого количества пользователей, поскольку серверные мощности ограничены.

Клиентский рендеринг предполагает вынесение логики обращения к базе данных на серверную часть, а обработку бизнес-логики и логики создания представления на клиентскую часть. Данный подход позволяет избежать вышеописанной проблемы с избыточной нагрузкой на сервер распределив ее между всеми клиентами. Однако в условиях данной архитектуры код бизнес-логики оказывается открытым, что значительно уменьшает возможности контроля целостности и безопасности данных, заставляя реализовывать ее только на стороне базы данных. При этом увеличивается нагрузка на клиентов, однако в условиях большого числа пользователей данный недостаток является скорее достоинством, поскольку разгрузка сервера значительно увеличивает потенциальное количество клиентов.» [15]

В статье «Сравнение классического процесса реализации веб-приложений и подхода с использованием библиотеки React» авторы делаю следующее заключение:

«Создать страницу с минимальным содержимым проще и быстрее с помощью классического подхода, так как не требуется ничего устанавливать. Это может быть удобно для создания небольших страниц. Однако данная

процедура совершается единожды за весь проект, поэтому такое преимущество незначительно при разработке крупных приложений.

Разработка и поддержка приложений с помощью React происходит значительно легче, чем при классическом подходе. Это осуществляется за счет встроенных в React механизмов.

По скорости отрисовки страниц также выигрывает способ разработки веб-приложений с использованием React. Загрузка такой страницы в среднем происходит в три раза быстрее, чем аналогичной, разработанной с помощью классического подхода.

Таким образом, так как веб-технологии стремительно развиваются, не стоит отказываться от новых библиотек и фреймворков. Они делают процесс разработки веб-приложений значительно проще и качественнее.» [4]

В статье «Анализ фреймворков для разработки современных веб-приложений» автор Чернецкий И. И. проводит сравнение характеристик самых популярных в настоящий момент UI-фреймворков Angular, Vue, React и Svelte, после чего приходит к выводу «важно понимать, что нет единственно правильного фреймворка для всех ситуаций. Выбор должен базироваться на конкретных требованиях проекта, опыте команды и долгосрочной стратегии развития. В то время как одни фреймворки могут предлагать высокую производительность и оптимизацию, другие выделяются гибкостью и легкостью использования. Также необходимо учитывать активность сообщества, качество документации и будущее развитие выбранного фреймворка.» [17]

В статье «Нестандартные архитектура в написание веб приложений» Яровая Е.В. описывает особенности UI-фреймворка Svelte, который отличается от большинства других способом преобразования исходного кода клиентской части веб-приложений в готовый код для работы в браузере.

«Принципы работы Svelte заключаются в том, что первым этапом мы пишем высокоуровневый, декларативный код, как и на остальных популярных фреймворках, далее в действие вступает компилятор, который превращает его

в банд с низкоуровневым и императивным кодом, с высокую производительность. Отсутствие виртуального дома, как не странно повышает производительность, так как изменение точечно, и мы точно знаем какой DOM узел изменился. В runtime остается только готовый самодостаточный код, по этой причине svelte называют исчезающим фреймворком, после сборки почти никаких следов не остается, что в свою очередь приводит к крайне небольшому конечному банду. К примеру, вендор старте svelte весить 3 Кбайт, а у Preact это почти 9 Кбайт значительная разница не так ли, а Preact, который является облегченной версией React. Это действительно новый подход к написанию веб приложений. Хранилище состояния в Svelte, это всего лишь обсервебал (объекты), на которые подписывает приложение и обновляет данные при подписке и все это подключается на этапе компиляции.» [18]

1.4 Обзор публикаций по теме оптимизации веб-приложений

Оптимизация веб-приложений обычно направлена на ускорение загрузки и отклика во время работы. Плохо, если веб-приложение будет медленно открываться и зависать во время работы, поэтому разработчики применяют различные способы оптимизации чтобы, приложение быстро загружалось и быстро реагировало на пользовательские действия.

В статье «Методы повышения производительности современных веб-приложений» описано несколько способов оптимизации веб-приложения, которые перечислены ниже.

Использование websocket в качестве канала связи для исключения множества цикличных запросов с целью получения обновленного состояния. Использование подготовки данных на серверной стороне для исключения передачи данных, в которых нет необходимости. В качестве примера правильного подхода приводится использование языка запросов graphql. Также рекомендуется использование на серверной стороне баз данных NoSQL для ускорения доступа к большим данным. [5]

В статье «Увеличение скорости загрузки web-приложений» предлагается несколько способов оптимизации, среди которых «кэширование данных на стороне сервера; кэширование web-страниц (на стороне сервера либо на стороне клиента); использование многоуровневой архитектуры FrontEnd-BackEnd; использование web-сервера, построенного по FSM (Finite State Machine), сжатие передаваемых данных средствами протокола HTTP. Все они в той или иной мере способствуют повышению производительности web-приложения и могут применяться как по отдельности, так и комбинироваться разработчиками web-приложений.» [9]

«Для уменьшения размеров файлов используют специальные утилиты (YUI Compressor, PngCrush, PngOptimizer). В файлах JavaScript и CSS удаляются лишние биты: удаляются пробелы, комментарии, переводы строк, заменяются на более короткие имена переменных. Изображения также оптимизируются. В результате объѐм файла уменьшается в среднем на 50-55%. Дополнительно на web-сервере файлы могут сжиматься утилитой GZIP. Суммарно такая обработка файлов может привести к уменьшению объема файлов на 80-85%.» [9]

В статье Методы оптимизации производительности web-приложений Логинова Н.В. перечисляет следующие методы оптимизации:

- «минификация и обфускация кода: минификация кода – это процесс удаления всех ненужных символов из исходного кода без изменения его функциональности. Обфускация – это метод, который делает код трудным для понимания и изменения, в то же время сохраняя его функциональность. Эти методы уменьшают размер файлов, что может привести к ускорению загрузки страниц.
- кэширование: это процесс хранения данных, которые используются повторно, для уменьшения времени, необходимого для загрузки или достижения этих данных. Кэширование может применяться на нескольких уровнях: на стороне клиента (в браузере), на уровне сервера или с использованием сетей доставки контента (CDN).

– оптимизация изображений: часто изображения являются наиболее тяжелыми для загрузки ресурсами на веб-странице. Их оптимизация может включать в себя изменение размеров, смену формата, сжатие без потери качества и др.
– использование CDN: CDN, или сети доставки контента, используются для быстрой доставки контента пользователям. Они позволяют хранить кэшированные версии контента на серверах, расположенных в разных местах мира, для ускорения загрузки контента.
– оптимизация базы данных: эффективность работы с базой данных зависит не только от самой СУБД, но и от того, как организована работа с ней. Важно правильно налаживать процессы индексации, проводить нормализацию и денормализацию данных, а также оптимизировать самые тяжелые и частые запросы.
– code Splitting: это метод разделения кода на отдельные пакеты, которые загружаются только при необходимости. Такой подход позволяет ускорить начальную загрузку страницы.
– использование браузерных асинхронных API и техники Lazy Loading: С асинхронными API и техникой отложенной загрузки можно выполнить некоторые задачи в фоновом режиме, пока клиент продолжает взаимодействовать с приложением.
– optimistic UI Updates: эта техника предполагает немедленное отображение действий пользователя в UI, даже если эти действия еще не подтверждены сервером. Такой подход может улучшить восприятие производительности приложения пользователем.» [8]

В связи с относительной новизной веб-приложений способы их оптимизации в целом, проблематика большого размера, а также способы создания легковесных приложений на сегодняшний день в научных ресурсах освещены довольно незначительно. Для более подробного изучения также были найдены и проанализированы статьи российских и зарубежных авторов.

В статье «Оптимизация веб-страницы: подробное руководство», несмотря на 2017 год ее написания, приведены способы оптимизации веб-страниц, которые актуальны и в настоящий момент для веб-приложений. Помимо уже перечисленных выше методов оптимизации также говорится о необходимости использования современных форматов для различных типов ресурсов, например: для изображений – webp, для шрифтов – woff2. Для изображений в векторном формате рекомендуется использование формата SVG. [10]

Также стоит дополнительно отметить наличие онлайн инструментов для удаления из файлов шрифтов символов, которые вряд ли пригодятся для конкретного проекта. Например, в проекте для русскоязычных пользователей вряд ли понадобятся символы, отличные от латиницы и кириллицы. В многоязычных проектах можно для каждой группы языков создавать отдельные файлы шрифтов, содержащих латиницу, общеупотребляемые символы и символы конкретного языка.

Вообще проблема медленной загрузки и зависания веб-приложений приобрела широкое распространение сравнительно недавно, а именно вместе с началом разработки больших приложений с использование UI-фреймворков, предоставляющих возможность создания одностраничных веб-приложений. Когда фреймворков еще не было, веб-приложения были многостраничными, то есть при переходе на новую страницу браузер очищал все ресурсы, которые использовала предыдущая страница и загружал новую. Это значит, что к каждой отдельной странице можно было подключить только необходимые дополнительные компоненты (стили, скрипты, медиаресурсы). Если на одной странице есть многофункциональная таблица, то стили и скрипт для ее работы подключался только на этой странице, на другой же странице подключались другие необходимые скрипты и стили.

В таблице 1 приведены обобщенные результаты по современному состоянию различных аспектов реализации клиентской части веб-приложений.

Таблица 1 - Современное состояние аспектов реализации клиентской части веб-приложений

Аспект	Современное состояние
Механизмы взаимодействия между серверной и клиентской частью	Сетевые соединения http и websocket.
Фреймворки и библиотеки для построения пользовательского интерфейса	React, Vue, Angular, Svelte, Backbone, Ember и другие.
Принципы реализации клиентской части веб-приложений	Одностраничное приложение и многостраничное приложение.
Сторона рендеринга разметки веб-страницы	Рендеринг на стороне сервера, рендеринг на стороне клиента.
Оптимизация трафика	GZIP сжатие, Content Delivery Network (CDN) - сети доставки контента.
Сокращение размера клиентской части веб-приложений	Минификация кода в процессе сборки файлов проекта, использование современных форматов изображений, видео, шрифтов и других ресурсов.

В современных веб-приложениях, построенных по принципу одностраничного приложения, как правило, однажды загруженный фрагмент кода для работы одного из компонентов остается загруженным и продолжает быть доступным даже, когда пользователь переходит на другие страницы. Таким образом, переходя по страницам, загружаются постепенно множество фрагментов кода, увеличивая потребление оперативной памяти.

В настоящее время сфера веб-разработки развивается довольно стремительно. Начавшись с небольших веб-сайтов, представленных набором статических веб-страниц, к настоящему времени веб-разработка по сложности и богатству функционала достигла уровня обычных приложений, устанавливаемых на персональные компьютеры. Уже сейчас есть веб-приложения для редактирования документов, таблиц, презентаций, макетов дизайна и множество других веб-сервисов, не уступающих по функционалу обычным устанавливаемым программам. И нужно заметить, что веб-сервисы приобретают все большую популярность, так как для их использования не нужно производить скачивание и установку программы на персональный компьютер.

Выводы по главе 1

В процессе проведения анализа научной литературы было выявлено более 30 источников, которые затрагивают архитектурные аспекты создания веб-приложений. Хорошо описаны механизмы взаимодействия между серверной и клиентской частью, разнообразие фреймворков для клиентской части веб-приложения, разные подходы создания клиентской части: одностраничное или многостраничное веб-приложений, приложения с рендерингом на стороне сервера и на стороне клиента. Все эти аспекты являются базовыми, они хорошо исследованы, общеизвестны и уже давно применяются в практической деятельности.

Тема оптимизации веб-приложений в научных публикациях освещена довольно слабо, но основные методы оптимизаций в найденных публикациях рассмотрены. В целом, оптимизация сводится к минимизации всех ресурсов и данных необходимых для работы веб-приложения и ускорение их получения с сервера, при чем ресурсы должны запрашиваться только в случае реальной необходимости.

Глава 2 Способы сокращения кода клиентской части веб-приложений

2.1 Минимизация подключаемых ресурсов веб-приложения

Сокращение размеров клиентской части веб-приложений как правило начинают с оптимизации размера всех подключаемых ресурсов. Сюда относятся стили, скрипты, шрифты, картинки, видео и другие медиаресурсы.

Css файлы, отвечающие за стилизацию веб-страниц, рекомендуется сжимать путем группировки правил и вырезки лишних пробелов. Браузер все равно поймет код в таком файле, но его размер станет немного меньше. Простейший пример минификации приведен на рисунке 6.

```
.block1{
 display:block;
}
.block2{
 display:block;
 height:32px;
}
.block3{
 display:block;
 height:32px;
 width:44px;
}
.block4{
 height:32px;
 width:44px;
}
```

```
.block1,.block2,.block3{display:block;}.block2,.block3,.block4{height
:32px;}.block3,.block4{width:44px;}
```

Рисунок 6 – Пример минификации простейшего css-кода

В примере на рисунке 6 объем исходного кода (слева) составляет 186 байта, а объем минифицированного кода (справа) составляет 104 байта (~56%). Таким образом есть возможность сокращать размер css-файлов на 30-60% в зависимости от содержания и повторяемости css-правил в конкретных файлах.

Минификация css-файлов обычно осуществляется на этапе сборки файлов проекта специальными модулями, такими как CssNano [22], CssClean [19] и другие. Помимо минификации во время сборки осуществляются и другие необходимые методы подготовки кода, например добавление css-правил с префиксами вендоров, удаление правил, которые не используются в коде html-разметки, если весь объем этого кода известен заранее.

Javascript-код также хорошо поддается минификации. В процессе минификации где это возможно имена переменных, функций, свойств и методов заменяются на более короткие комбинации из 1-2 символов, вырезаются пробелы, преобразуются конструкции кода в более короткие без нарушения логики работы программы. В примере на рисунке 7 объем исходного кода (слева) составляет 353 байта, а объем минифицированного кода (справа) составляет 107 байтов (~30%). Согласно бенчмаркам минификаторов javascript кода популярных библиотек [25], после минификации код получается размеров 30-35% от исходного, соответственно экономия составляет 65-70%. Для минификации используются такие инструменты, как uglify-js [34], Terser [13], Closure Compiler [20] и другие.

```
(function () {
  function sumary(firstArg, secondArg, ...otherArgs) {
    let sum = firstArg + secondArg;
    otherArgs.forEach((arg) => {
      sum += arg;
    });
    return sum;
  }

  const deltaX = 10;
  const deltaY = 5;
  const deltaZ = 46;
  const deltaSum = sumary(deltaX, deltaY, deltaZ);
  console.log(deltaSum);
})();
```

```
!function(){const n=function(n,o,...c){let t=n+o;return
c.forEach(n=>{t+=n}),t}(10,5,46);console.log(n)}();
```

Рисунок 7 – Пример минификации простейшего javascript-кода

Что касается шрифтов, то самый минимальный с точки зрения объема загружаемых файлов вариант – это использование системного шрифта, который используется по умолчанию в операционной системе. Конечно, в разных операционных системах в качестве системного используются разные шрифты и это нужно учитывать в случае его использования. Если же при разработке проекта все же решили использовать не системный шрифт, а отдельный, который подключается через отдельные файлы шрифтов, то стоит использовать современный формат, в 2024 году таковым является woff2. На рисунке 8 представлено по 4 файла разного формата двух разных шрифтов, скачанных в онлайн-сервисе Google Fonts [12].

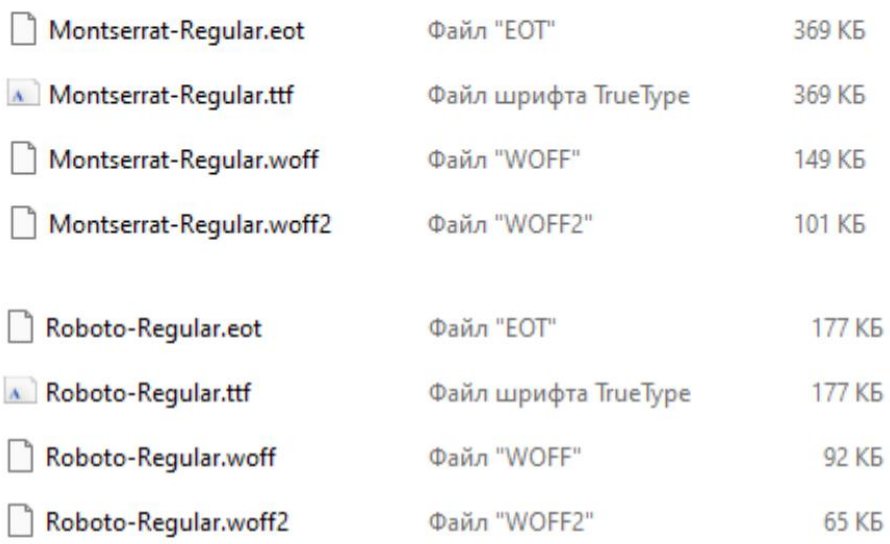

Рисунок 8 – Сравнение размеров файлов шрифтов

Как видим, woff2 имеет размер примерно на 30% меньше, чем woff и на 70% меньше, чем ttf. Помимо использования современного формата woff2, можно очистить файлы шрифта от символов, которые вряд ли будут использоваться в определенном проекте. Для фильтрации символов в шрифте можно воспользоваться онлайн-сервисом Transfonter [14]. Например, для русскоязычного проекта можно ограничиться набором символов латиницы, кириллицы, цифр и необходимых спецсимволов.

После удаления лишних символов из шрифта Roboto, размеры файлов которого представлены на рисунке 9, получаем гораздо меньшие размеры файлов.

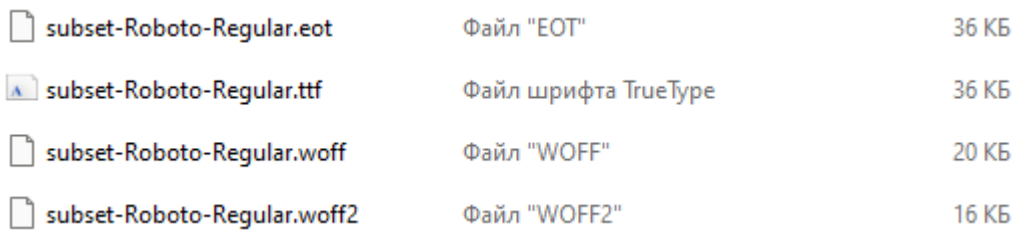

Рисунок 9 – Сравнение размеров файлов шрифта Roboto, очищенного от лишних символов

Как видим, после удаления лишних символов из шрифта Roboto, скачанного на сайте сервиса Google Fonts, размер файла woff2 сократился на 75%, с 65 до 16 Кбайт.

При использовании картинок на веб-сайтах и в веб-приложениях следует учитывать размер мест, где они отображаются. Например, в место с размером 100x100 пикселей нецелесообразно грузить картинку с размером 800x800 пикселей, так как размер файла будет при этом гораздо больше, а преимуществ от этого пользователь никаких не получит, так как браузер масштабирует полученный файл до размеров места вывода. Что же касается форматов файлов изображений, то стоит использовать наиболее современные форматы, поддерживаемые большинством браузеров. На момент 2024 года для растровых изображений оптимальным считается формат webp, а для векторных – svg.

В ходе исследования по сравнению размеров файла jpg и webp было сделано заключение, что «средний размер файла WebP на 25–34 % меньше по сравнению с размером файла JPEG с эквивалентным индексом SSIM. Эти результаты показывают, что WebP может обеспечить значительные улучшения сжатия по сравнению с JPEG.» [6]

При размещении видеороликов и аудиофайлов на веб-страницах желательно давать пользователю возможность выбирать качество на его усмотрение, так как при медленном интернете видео и аудио высокого качества могут долго грузиться.

2.2 Отложенная загрузка ресурсов веб-приложения

Страницы веб-приложений могут быть довольно длинными. Для начала взаимодействия со страницей желательно побыстрее загрузить медиа ресурсы (картинки, видео, аудио) верхней ее части, а далее можно загружать необходимый медиа ресурсы по мере того, как пользователь пролистывает страницу вниз, а в случае с видео и аудио можно вообще начинать загрузку только если пользователь нажал на кнопку запуска воспроизведения.

Для отложенной подгрузки изображений и встроенных страниц, которые могут содержать различный медиа контент, можно использовать атрибут loading=lazy у тегов img и iframe [26], а для тегов video можно использовать атрибут preload=none.

С появлением фреймворков для клиентской части веб-приложений, реализуемых в парадигме одностраничного приложения, резко увеличился объем javascript кода, вследствие чего появилась необходимость в отложенной загрузке фрагментов исполняемого кода, так как если весь исходный код скриптов собрать в единый файл, то он будет слишком большого размера, что негативно отразится на времени первоначальной загрузки веб-приложения.

Для разделения исполняемого кода скриптов клиентской части веб-приложений с возможностью последующей их отложенной загрузки при необходимости в большинстве сборщиков javascript кода уже существуют встроенные возможности. Обычно они обозначаются термином «Динамический импорт». Такую возможность поддерживают инструменты сборки Webpack, Vite, Rollup и другие. Проанализировав объем первоначально загружаемых javascript файлов и объем загружаемых javascript файлов по мере

перехода по разным страницам на 3-х веб-ресурсах, работающих по схеме одностраничного приложения, было выявлено, что первоначально грузится примерно половина всего объема javascript файлов, вторая половина получается браузером постепенно в процессе перехода по разным страницам. В итоге при первоначальной загрузке получается сэкономить около 50% на объеме загрузки файлов скриптов.

При использовании сборщиков javascript кода, если стоит задача максимально сократить размеры файлов для браузера, то нужно иметь в виду, что разные инструменты собирают код с некоторыми особенностями внутренней его компоновки, особенно если используются средства динамического импорта. Эти особенности компоновки кода из сотен или даже тысяч файлов с исходным кодом также могут влиять на размер итогового файла или нескольких файлов, которые сборщик готовит для использования в браузере.

Обобщенно технологии и инструменты оптимизации размера ресурсов клиентской части веб-приложений приведены в таблице 2.

Таблица 2 - Технологии и инструменты оптимизации размера ресурсов клиентской части веб-приложений

Вид оптимизация	Технологии и инструменты
Минификация javascript	Uglify-js, Terser, Closure Compiler и другие
Минификация css	CssNano, CssClean и другие
Оптимизация размера файлов шрифтов	Онлайн-сервис Transfonter
Отложенная загрузка картинок и встроенных страниц в браузере	Атрибут loading=lazy у тегов img и iframe
Отложенная загрузка видео контента в браузере	Атрибут preload=none у тега video
Отложенная загрузка скриптов и стилей	Механизм динамического импорта в инструментах сборки Webpack, Vite, Rollup

Для отложенной подгрузки css файлов можно подготавливать отдельные файлы для конкретных страниц и подключать их только во время нахождения пользователя на конкретной странице. Альтернативным путем для отложенной загрузки css-стилей является использование инструментов css-in-

js, суть которой заключается в хранение css стилей внутри js кода. Соответственно, если используется динамическая подгрузка javascript файлов, то вместе с ней динамически подгружается и фрагмент css правил.

Для минификации и отложенной подгрузки стилей и скриптов существует по несколько альтернативных инструментов, имеющих свои особенности. Сфера веб-разработки довольно динамична. Каждый год появляются новые инструменты, способствующие улучшению процесса разработки и оптимизации различных ее аспектов, вытесняя при этом более старые инструменты.

Выводы по главе 2

Для сокращения размера веб-приложения необходимо минимизировать все составляющие его ресурсы: html, css, javascript, картинки, видео, шрифты и так далее.

Для минификации javascript можно использовать uglify-js [34], Terser [13], Closure Compiler [20] и другие.

Для минификации css используются инструменты CssNano [22], CssClean [19] и другие.

Для сокращения размера шрифтов, включая фильтрации символов в шрифте, можно воспользоваться онлайн-сервисом Transfonter [14].

Для отложенной загрузки медиа ресурсов в браузере есть специальные атрибуты, loading=lazy у тегов img и iframe [26], а для тегов video можно использовать атрибут preload=none.

Для отложенной загрузки скриптов и стилей существует механизм динамического импорта, которые поддерживается многими популярными инструментами сборки javascript кода, такими как Webpack, Vite, Rollup.

Глава 3 Модель веб-приложения с размещением логики пользовательского интерфейса в серверной части

Методы оптимизации размера клиентской части веб-приложений, рассмотренных в разделах 2.1 и 2.2 довольно широко применяются в настоящее время, но несмотря на это, их размер остается довольно внушительным из-за огромного количества логики отображения и взаимодействия с пользователем. Это обуславливает медленную начальную загрузку и большой расход оперативной памяти. Это касается только веб-приложений, работающих по принципу одностраничного приложения (SPA – single page application), при котором страница получается от сервера только при первом заходе, а при всех последующих переходах по страницам, приложение с помощью AJAX запросов получает данные с сервера и обновляет разметку на странице. В случае же многостраничных веб-приложений такой проблемы обычно не возникает, так как на странице подключаются скрипты только тех компонентов, которые там используются.

В случае использования одностраничных веб-приложения большого размера на персональном компьютере или на высокопроизводительных мобильных устройствах с высокоскоростным интернетом, проблемы вряд ли возникнут, но на низкопроизводительных мобильных устройствах веб-приложения большого размера могут подвисать или при низкой скорости интернета будут медленно загружаться.

В современных веб-приложениях, разработанных с применениями фреймворков, таких как react, vue, angular и других, им подобных, часто проблема возникает именно с объемом javascript кода, в котором находится логика отображения и взаимодействия с пользователем. Даже с использованием разделения и отложенной загрузки фрагментов кода при необходимости, который был рассмотрен в разделе 2.2, после просмотра множества страниц существенная часть кода скриптов оказывается загруженной и содержится в оперативной памяти.

3.1 Теоретическое представление модели размещения логики пользовательского интерфейса в серверной части веб-приложения

Для минимизации javascript кода клиентской части веб-приложения можно перенести логику пользовательского интерфейса из клиентской части веб-приложения в серверную часть, а в процессе работы передавать ее через ajax-запросы по мере необходимости на клиентскую часть. Как правило, такая необходимость возникает будет возникать при переходе на другую страницу либо при открытии модальных (всплывающих) окон, объем контента в которых может быть сопоставим с контентом обычной веб страницы. Упрощенная схема взаимодействия представлена на рисунке 10.

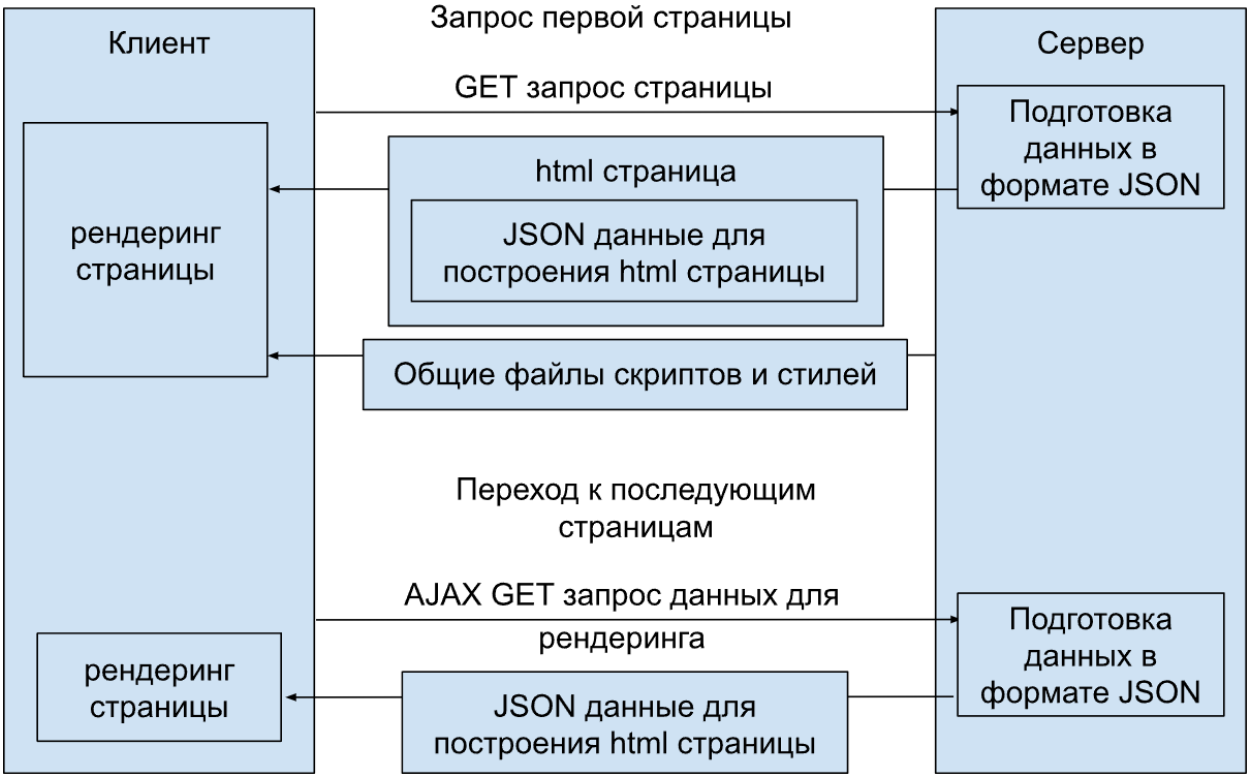

Рисунок 10 – Модель веб-приложения с размещением логики пользовательского интерфейса в серверной части

Альтернативным вариантом взаимодействия будет вариант, представленный на рисунке 11, при котором начальная html страница не

содержит данные для рендеринга, а они получаются отдельным ajax-запросом, как и в случае перехода на последующие страницы. Но такой вариант увеличит время запуска веб-приложения.

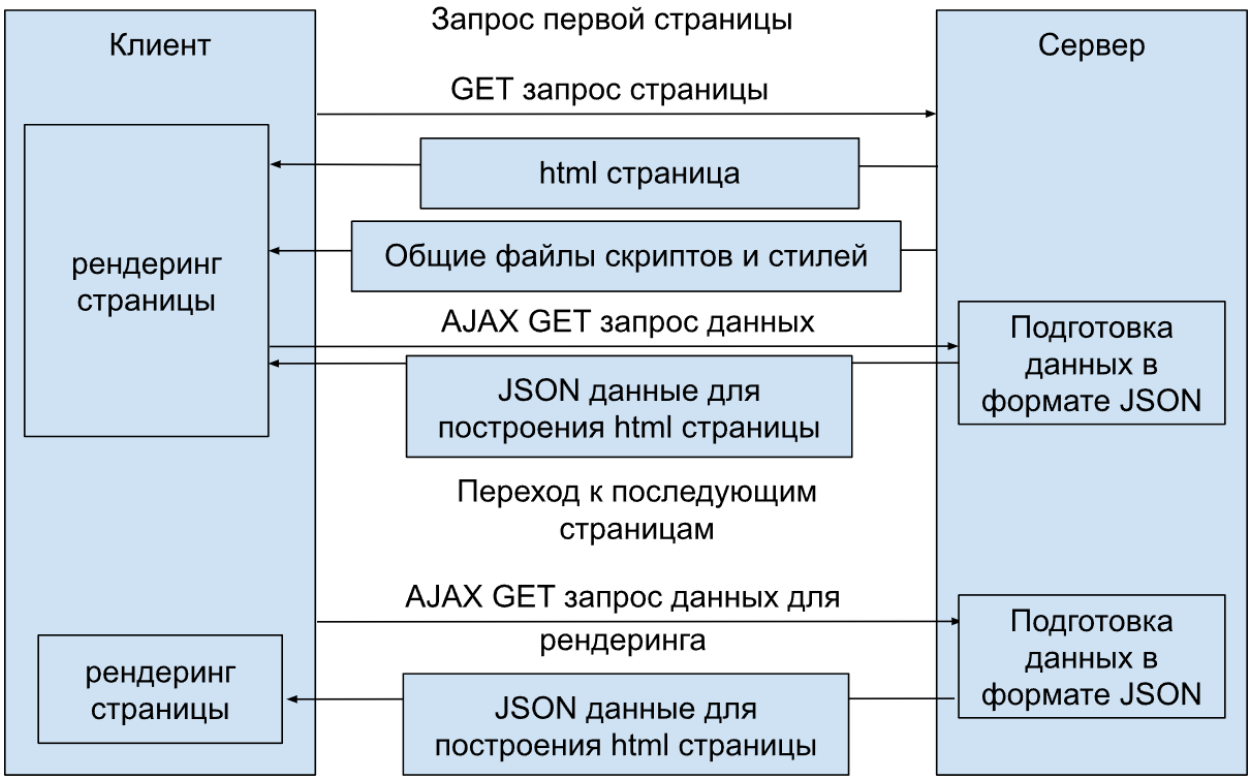

Рисунок 11 – Альтернативная схема взаимодействия серверной и клиентской частей веб-приложения при переходе к новой странице

На текущий момент для передачи структурированных данных можно воспользоваться форматом JSON, который сейчас поддерживается в большинстве языков программирования.

«JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом. Несмотря на происхождение от JavaScript (точнее, от подмножества языка стандарта ECMA-262 1999 года), формат считается независимым от языка и может использоваться практически с любым языком

программирования. Для многих языков существует готовый код для создания и обработки данных в формате JSON.» [24]

Используя формат JSON можно легко передавать структуру разметки конкретной html-страницы или структуру веб-формы. На рисунках ниже представлены примеры данных в формате JSON для построения элементов пользовательского интерфейса, которые могут передаваться с сервера при необходимости и на основе которых клиентская часть веб-приложения будет строить веб-страницу, которая отображается пользователю в браузере.

При использовании первой схемы взаимодействия при заходе на страницу веб-приложения браузер отправит обычный GET запрос, в ответ на который получит пустую страницу с подключенными файлами стилей и скриптов, а также данные для рендеринга контента страницы в формате JSON, вставленные в отдельный тег script. Пример разметки такой начальной страницы приведен на рисунке 12.

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="UTF-8" />
    <title>Web application</title>
  </head>
  <body>
    <div id="appRoot"></div>
    <script>
      const pageScheme = {
        elem: 'block',
        class: 'page',
        children: [
          {
            elem: 'block',
            class: 'page__header',
            children: [/* ... */],
          },
          // ....
        ],
      };
    </script>
    <script src="/script.js"></script>
  </body>
</html>
```

Рисунок 12 – Содержимое html файла начальной страницы веб-приложения

После получения начальной страницы браузер еще загрузит подключенные к странице файлы скриптов и стилей. Далее запустится скрипт, который используя имеющиеся на странице данные для рендеринга, создаст разметку и вставит ее на страницу, после чего веб-приложение будет готово для взаимодействия с пользователем.

При нажатии на ссылку или при другом виде взаимодействия, которое должно привести к переходу на другую страницу, приложение отправит ajax-запрос на сервер за данными для рендеринга необходимой страницы, после получения которой отрендерит новую разметку и обновит ею контент страницы.

Таким образом, клиентская часть веб-приложения является просто набором готовых для использования компонентов, которые будут использоваться функцией рендеринга для генерации контента страниц веб-приложения. Наглядно структура показана на рисунке 13.

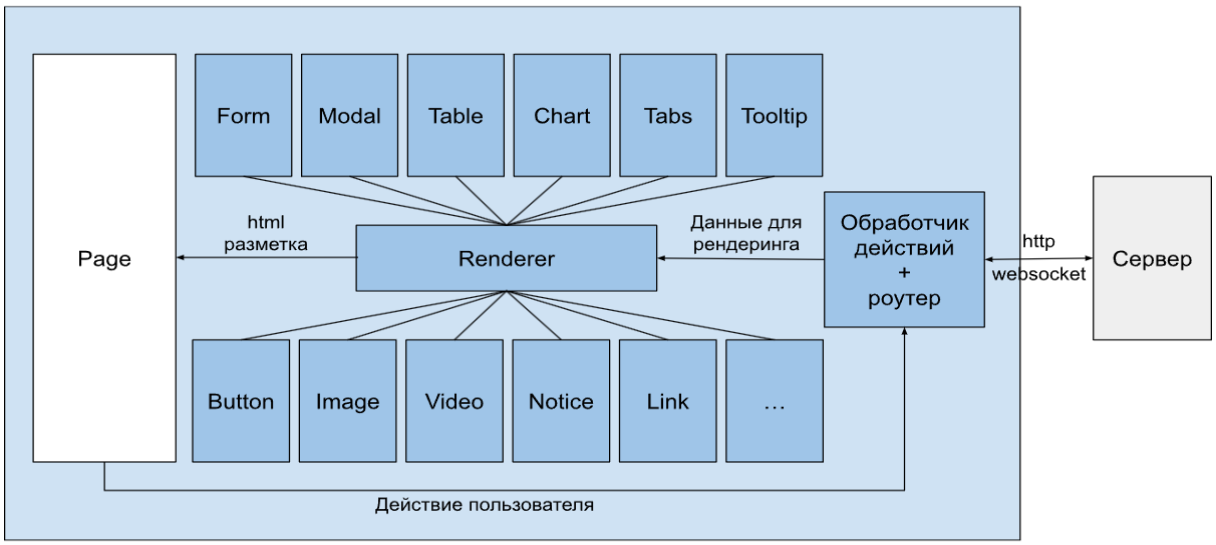

Рисунок 13 – Структура веб-приложения

Компоненты могут вкладываться внутрь друг друга. Например, на странице могут быть табы (Tabs), в одном из табов может размещаться форма

(Form), в форме будет подсказка (Tooltip) к одному из полей, а подсказке будет ссылка (Link) на страницу со справочной информацией.

Для передачи структуры веб-страницы, которая в браузере представляется набором вложенных друг в друга html-тегов, можно использовать схему данных, представленный на рисунке 14.

```
{
  "elem": "block",
  "class": "page",
  "children": [
    {
      "elem": "block",
      "class": "page__header",
      "children": [
        { "elem": "@Logo" },
        { "elem": "@Search" },
        { "elem": "@ProfileLink" }
      ]
    },
    {
      "elem": "block",
      "class": "page__content",
      "children": [
        {
          "elem": "h1",
          "class": "page__title",
          "children": "Форма регистрации"
        },
        {"elem": "@Form"...}
      ]
    },
    {
      "elem": "block",
      "class": "page__footer",
      "children": [
        { "elem": "@Logo" },
        {
          "elem": "block",
          "class": "page__copyright"
        }
      ]
    }
  ]
}
```

Рисунок 14 – Данные общей структуры страницы в формате JSON

Действия пользователя (клик, скролл, перетаскивание и так далее) могут обрабатываться компонентами, в которых они произошли. Например, пользователь переключает табы, заполняет форму, закрывает всплывающее уведомление и так далее. Также можно добавлять глобальные обработчики, которые будут обрабатывать переход по ссылкам, отправлять статистическую информацию о действиях пользователя на странице, обрабатывать ошибки и так далее.

Как видно из приведенного примера на рисунке 14, есть несколько разных elem (элемент): block, text, h1 и другие элементы, названия которых начинаются с символа @. Block – простейший блочный элемент на всю ширину родительского элемента. Text – строчный элемент с текстовым содержимым. H1 – заголовок первого уровня. @Logo, @Search, @ProfileLink, @Form, @Link – это базовые компоненты клиентской части приложения, которые имеют некоторую внутреннюю логику отображения и/или взаимодействия. Символ @ в начале как раз и будет подсказывать механизму рендеринга, что это не обычный html-тег, а компонент с имеющейся внутренней логикой, написанной разработчиком клиентской части веб-приложения.

Также легко можно представить схему данных для веб-формы, пример которой представлен на рисунке 15.

Реализуя такой подход, львиная часть бизнес-логики, отвечающей за представление, которая зачастую размещается в клиентской части приложения, будет генерироваться на серверной стороне, но несмотря на это, те компоненты, которые нуждаются в сложной интерактивной логике, реализуют интерактивность именно на клиентской стороне. Эта реализация общей интерактивности обычно вообще не связана с нюансами конкретной бизнес-логики, отвечающей за отображение и взаимодействие. Например, данные для полей формы содержат параметры валидации (допустимое количество символов в поле или маска ввода), это часть бизнес-логики, и она

будет приходить с серверной части приложения. А сам механизм валидации, общий для всех форм, будет реализован на клиентской стороне.

```
{
  "elem": "@Form",
  "url": "/reg/",
  "method": "POST",
  "items": [
    {
      "type": "text",
      "name": "login",
      "required": true,
      "validations": [
        { "type": "minLength", "arg": 3 },
        { "type": "maxLength", "arg": 64 }
      ]
    },
    {
      "type": "password",
      "name": "password",
      "required": true,
      "validations": [
        { "type": "minLength", "arg": 8 },
        { "type": "maxLength", "arg": 32 }
      ]
    },
    {
      "type": "content",
      "children": [
        {"elem": "text", "children": "Я принимаю "},
        {"elem":  "@Link", "url": "/rules/", "children": "условия пользования сайтом"}
      ]
    },
    {
      "type": "submit",
      "text": "Зарегистрироваться"
    }
  ]
}
```

Рисунок 15 – Данные для элемента формы регистрации в формате JSON

В случае необходимости реализации некой логики взаимодействия, выходящей за пределы стандартной функциональности имеющихся компонентов, можно просто добавить отдельный компонент, реализующий дополнительную логику в клиентской части приложения и использовать его там, где нужно.

Например, у нас есть стандартный компонент формы, но нам помимо функционала самой формы с заполнением полей, нужно, чтобы в форме редактирования данных о товаре на основе введенных данных подсчитывался и выводился рейтинг заполненности информации по сложным формулам с множеством условий. Причем заказчик функционала хочет, чтобы этот подсчет реализовывался именно в клиентской части для мгновенного пересчета и отображения.

Для реализации задуманного мы можем создать компонент с недостающим нам функционалом, то есть компонент, который на основе данных из полей формы будет вычислять и выводить рейтинг. Пример кода такого компонента представлен на рисунке 16. Конечно, должна быть реализована и возможность подключения такого компонента внутри компонента формы с передачей ему данных состояния полей формы.

```
09  export function ProductRating({ fieldsState }) {
10    const rating = fieldsState.title.length +
11      fieldsState.description.length / 10 +
12      fieldsState.attributes.length * 10;
13
14    return <div className="productRating">{rating}</div>;
```

Рисунок 16 – Пример кода отдельного компонента

Таким образом можно реализовать логику пользовательского интерфейса, которую не целесообразно добавлять в общий функционал. Как правило, таких особых случаев, где требуется что-то эксклюзивное в определенных местах, довольно мало, но в рамках предлагаемого подхода сложностей с реализацией возникать не должно.

Используя представленный подход, даже если в нашем приложении будет тысяча различных страниц, форм, таблиц и других довольно объемных по содержанию элементов, код клиентской части приложения будет гораздо

меньше, чем при широко распространенном подходе, когда вся логика представления хранится в клиентской части приложений. И хотя вышеописанный пример касается в большей степени веб-приложений, но аналогичный подход можно использовать и при разработке мобильных приложений. В этом случае размер мобильного приложения сократится, а также не потребуется реализовывать логику представления отдельно в клиентской части мобильной версии веб-приложения и отдельно в коде мобильного приложения.

Помимо сокращения размера javascript файлов, указанный подход имеет преимущество и по части трудозатратности процесса разработки. Он позволит в некоторой мере сократить затраты времени отдела разработки в случае, когда серверная и клиентская части разрабатываются разными сотрудниками. Как правило, серверная часть разрабатывается backend-разработчиком с использованием языков программирования PHP, Java, Go, Python и др. Клиентская часть разрабатывается фронтенд-разработчиком с использованием языка программирования JavaScript.

Сокращение времени разработки станет возможным благодаря размещению бизнес-логики представления в серверной части. Таким образом, сократится время участия фронтенд разработчика, а по ряду задач, в его привлечении вообще не будет необходимости. Разработчик серверной части ознакомится с техническим заданием, вникнет в задачу и реализует ее самостоятельно, используя для отображения уже имеющиеся стандартизированные компоненты в клиентской части, конфигурация для которых будет передана из серверной части веб-приложения.

Так как логика пользовательского интерфейса сосредоточена в серверной части, то путем анализа javascript кода клиентской части невозможно узнать, какие веб-адреса могут быть в приложении, какой контент будет выводиться на них. Возможно, этот пункт будет преимуществом для приложений с разным уровнем доступа, где имеется контент, доступный только узкому кругу пользователей.

Учитывая тот факт, что логика хранится или генерируется на сервере в формате структурированных данных, то это подразумевает использование декларативного подхода при разработке этой логики. Декларативный подход предполагает описание конечного результата, а не последовательности действий для его достижения, поэтому такой код более чистый и читабельный, более легок в поддержке и имеет меньшую подверженность появлению ошибок.

К недостаткам предлагаемой модели можно отнести следующее: увеличение объема кода серверной части приложения на величину логики пользовательского интерфейса; незначительное увеличение серверной нагрузки, так как потребуется не просто отдать данные, а подготовить на их основе структуру пользовательского интерфейса.

3.2 Пример реализации клиентской части веб-приложения с использованием модели размещения логики пользовательского интерфейса в серверной части веб-приложения

Для проверки работоспособности модели размещения логики пользовательского интерфейса в серверной части веб-приложения разработаем демонстрационное приложение, включающее набор необходимых компонентов, характерных для корпоративных веб-приложений. Для разработки серверной части будем использовать язык программирования PHP, а для разработки клиентской части будем использовать язык программирования javascript. Клиентскую часть реализуем по принципу одностраничного веб-приложения с использованием библиотеки react [31].

React – это библиотека для построения пользовательских интерфейсов с применением компонентного подхода. В современных веб-приложениях помимо самой библиотеки react используются также и дополнительные. Например, react-router для управления навигацией по приложению. Redux,

recoil, mobx – для управления состоянием веб-приложения. Но так как наше веб-приложение подразумевает хранение всей логики пользовательского интерфейса на серверной стороне, то нам ни роутинг, ни глобальное состояние на стороне клиента не понадобится.

В дополнение к библиотеке react, будут использоваться отдельные компоненты из библиотеки готовых компонентов Mui Material [28]. Конечно, любые компоненты можно разрабатывать и самостоятельно, но сейчас в этом нет смысла, так как уже разработаны и имеются в открытом доступе сотни готовых, протестированных компонентов, использующихся в тысячах разнообразных проектов, включая коммерческие.

Работа приложения начинается с начальной html страницы, которая отправляется с сервера в браузер при переходе пользователя по адресу веб-приложения. Начальная страница не содержит контента для отображения пользователю, но содержит подключение необходимых ресурсов, а именно ссылки на файлы скриптов и стилей. Пример кода начальной страницы показан на рисунке 17.

```
<!doctype html>
<html lang="ru" class="page">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>App</title>

  <script>
    window._initialContent = <?php echo json_encode($initialData ?? '{}'); ?>
  </script>

  <link rel="stylesheet" href="/assets/style.css">
</head>
<body>
<div id="root" class="page__root"></div>
<script src="/assets/main.js" type="module"></script>
</body>
</html>
```

Рисунок 17 – Html разметка начальной страницы

В нашем случае подключается 1 файл стилей /assets/style.css, и 1 файл скриптов /assets/main.js. Также на странице присутствует встроенный скрипт,

в котором выполняется сохранение в переменную window._initialContent данных для рендеринга текущей страницы. В данном случае текущая страница является первой, на которую зашел пользователь, поэтому данные должны содержать структуру интерфейса не только контентной части страницы, но и структуру тех частей веб-страницы, который не меняются на всех страницах, например шапка (header), боковая панель (sidebar), подвал (footer). Стандартная структура веб-страницы отображена на рисунке 18.

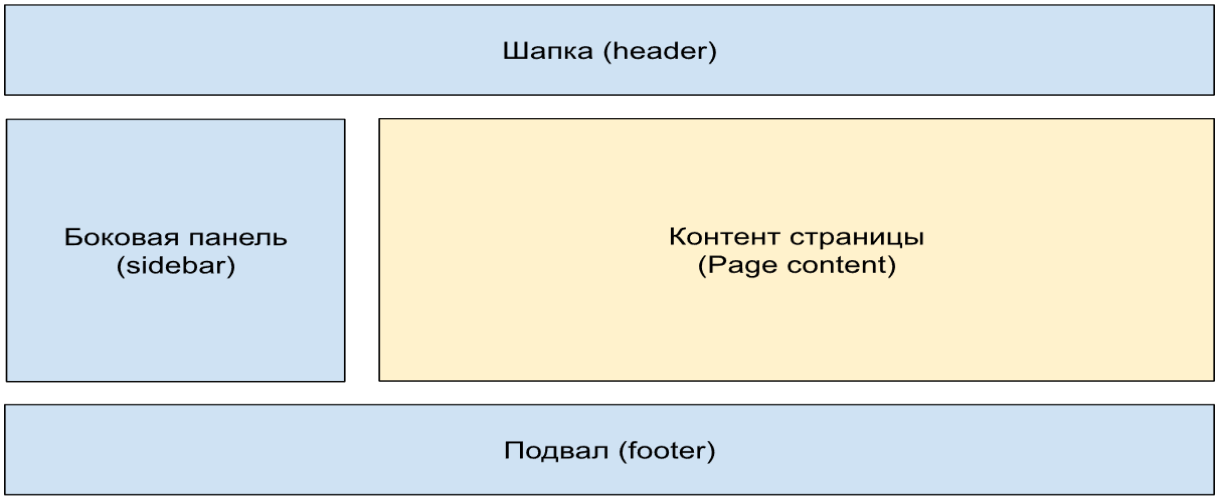

Рисунок 18 – Обычная структура веб-страницы

В демонстрационном веб-приложении нет необходимости в размещении боковой панели, поэтому вместе с структурой контента страницы (content) присутствуют данные по структуре шапки (headerContent) и подвала (footerContent), как видно на рисунке 19.

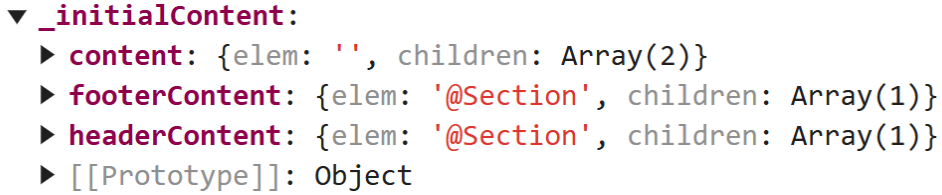

Рисунок 19 – Состав переменной window._initialContent

Шапка (header) и футер (footer) будут отрендерены при запуске веб-приложения, далее при переходах между страницами не предполагается изменение их структуры. Но даже при возникновении необходимости их изменения на отдельных страницах, можно для таких страниц вместе с данными структуры контентной части страницы передавать и структуру других частей. Рассмотрим более подробно состав и назначение ключевых частей демонстрационного веб-приложения, представленных на рисунке 20.

```
43  export default function App({ initialContent }) {
44    const [pageContent, setPageContent] = useState(initialContent.mainContent)
45    return (
46      <RendererProvider components={components}>
47        <NotificationsProvider maxSnack={4}>
48          <ModalsProvider>
49            <Page
50              mainContent={pageContent}
51              footerContent={initialContent.footerContent}
52              headerContent={initialContent.headerContent}
53            />
54
55            <UiHandler setPageContent={setPageContent} />
56          </ModalsProvider>
57        </NotificationsProvider>
58      </RendererProvider>
59    );
60  }
```

Рисунок 20 – Код файла App.jsx

RendererProvider – получает набор компонентов (components) и предоставляет любому компоненту доступ к возможности отрендерить фрагмент интерфейса на основе его структуры в формате JSON.

NotificationsProvider – предоставляет возможность любому компоненту вывести небольшое уведомление об успешном действии или о возникновении ошибки.

ModalsProvider – предоставляет удобный доступ любому компоненту веб-приложения к открытию модальных окно по центру страницы.

Page – компонент, задающий структуру основных компонентов страницы, в нашем случае – header, content, footer. Код компонента Page показан на рисунке 21.

```jsx
import ...

export default function Page({ mainContent, headerContent, footerContent }) {
  const renderFromJson = useRenderer();

  return (
    <>
      <div className="page__header">{renderFromJson(headerContent)}</div>
      <div className="page__content">{renderFromJson(mainContent)}</div>
      <div className="page__footer">{renderFromJson(footerContent)}</div>
    </>
  );
}
```

Рисунок 21 – Код файла Page.jsx

UiHandler – глобальный обработчик действий пользователя, основным из которых являются клики. Клик по элементам, которые не обрабатывается внутри какого-либо компонента, могут быть обработаны глобальным обработчиком. Всего может быть два варианта обработки клика: обычная ссылка и кликабельный элемент (ссылка без адреса или кнопка) со специальным атрибутом data-action, как показано на рисунке 22.

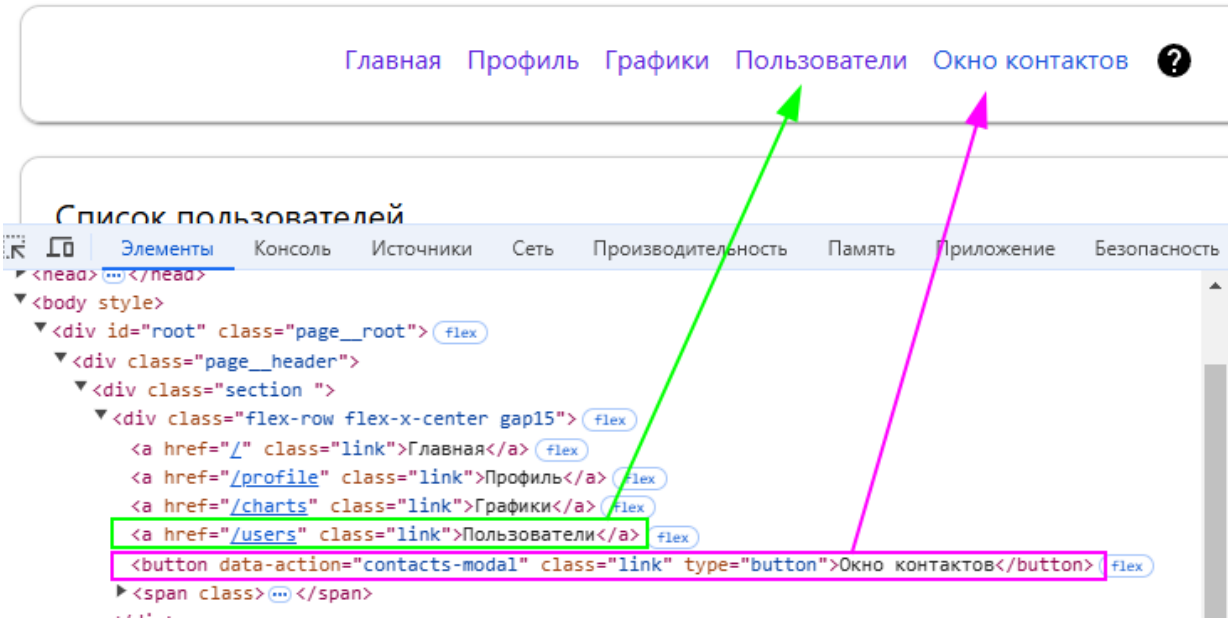

Рисунок 22 – Обычная ссылка и элемент с действием contacts-modal

Если это обычный ссылка, представленная тегом , которая предполагает переход на другую страницу, производим изменение URL адреса и отправляем запрос /api/page-content/?url=... за данными для рендеринга новой страницы, после прихода ответа – рендерим новую страницу.

Если у элемента есть некий специальный атрибут, например пусть это будет data-action=..., то действие пользователя будет обработано функцией sendElementAction, код которой представлен на рисунке 23. Функция отправляет запрос на сервер по адресу /api/action/ с передачей в теле запроса названия и параметров действия. Сервер обработает действие и в ответ может прислать:

– тип и текст уведомления, которое необходимо вывести;
– данные для рендеринга и вывода в модальном окне;
– адрес новой страницы, на которую необходимо перейти.

```
14    const sendElementAction = (node) => {
15      const actionName = node.dataset.action;
16      const actionData = node.dataset.actionData ?? "";
17
18      POST("/api/action/", { actionName, actionData }).then((data) => {
19        // если нужно перейти к новой странице
20        if (data.newLocation) {
21          history.pushState(null, "", data.newLocation);
22          loadPage(data.newLocation);
23        }
24
25        // если нужно открыть модальное окно
26        if (data.modal) {
27          openModal(data.modal);
28        }
29
30        // если нужно вывести короткое уведомление
31        if (data.notification) {
32          showNotification(data.notification.content, {
33            variant: data.notification.type,
34          });
35        }
36      });
37    };
```

Рисунок 23 – Код функции sendElementAction

Далее рассмотрим механизм рендеринга. Выше уже было сказано, что возможность рендеринга предоставляется компонентом RendererProvider. Он принимает набор необходимых компонентов, и предоставляет функцию рендеринга.

Возможность рендеринга фрагмента разметки на основе данных в формате JSON может быть получена либо с помощью использования отдельного компонента Renderer, как показано на рисунке 24, либо с помощью хука useRenderer (только в функциональных компонентах), как показано на рисунке 25.

```
import ...

export class FormCheckbox extends FormFieldBase {
  onChange = (e) => {
    this.props.onChange(this.props.name, e.target.checked);
  };

  render() {
    const { name, value, label } = this.props;

    return (
      <FormControlLabel
        control={<Checkbox name={name} checked={value || false} onChange={this.onChange} />}
        label={<Renderer content={label} />}
      />
    );
  }
}
```

Рисунок 24 – Использование компонента Renderer

```
import React from "react";
import { default as MuiTooltip } from "@mui/material/Tooltip";

import { useRenderer } from "../Renderer";

export default function Tooltip({ children, tooltip }) {
  const renderFromJson = useRenderer();

  return (
    <MuiTooltip title={renderFromJson(tooltip)} arrow>
      {renderFromJson(children)}
    </MuiTooltip>
  );
}
```

Рисунок 25 – Использование хука useRenderer

Демонстрационное веб-приложение содержит следующие компоненты:

– Img, Link, Button, Section – самописные простейшие компоненты.
– Form – компонент собственной реализации, который включает в себя компоненты ToggleButton, Slider, Select, RadioGroup, TextInput, DatePicker, Checkbox из библиотеки Mui Material.
– Icon, Tabs, Accordion, Tooltip, Table, Chart, DataTable, Dialog, Snackbar – отдельные компоненты из библиотеки Mui Material.

Для всех компонентов из сторонней библиотеки реализованы компоненты обертки, чтобы контролировать передаваемые параметры, а также реализовать возможность рендеринга разметки на основе данных в формате JSON, где это необходимо. На рисунке 26 мы можем увидеть пример такой обертки для компонента Tooltip (подсказка). Прием параметров ограничен двумя свойствами: children и tooltip. Для обоих параметров реализована возможность рендеринга контента на основе JSON данных. Данные для отображения такого компонента на веб-странице представлены на рисунке 26 и результат рендеринга представлены на рисунке 27.

```
{
  "elem": "@Tooltip",
  "tooltip": {
    "elem": "",
    "children": [
      "Это демонстрационное веб-приложение, в котором вся логика пользовательс...",
      { "elem": "br" },
      "Можете посмотреть графики ",
      {
        "elem": "@Link",
        "url": "/charts",
        "children": ["здесь"]
      }
    ]
  },
  "children": [
    {
      "elem": "@Icon",
      "type": "help"
    }
  ]
}
```

Рисунок 26 – Данные для рендеринга в формате JSON

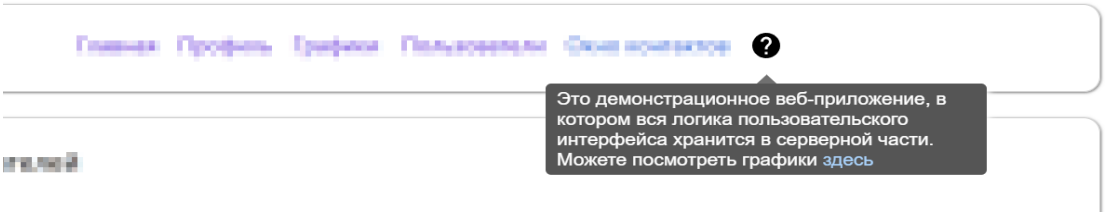

Рисунок 27 – Результат рендеринг компонента Tooltip

Далее подробно разберем функцию renderFromJson, которая непосредственно рендерит дерево компонентов на основе JSON схемы. Код функции renderFromJson показан на рисунке 28.

```
export function renderFromJson(item, uniqKey = "", components = {}) {
  if (typeof item === "string" || typeof item === "number") {
    return item;
  }

  if (item instanceof Array) {
    return item.map((contentItem, index) =>
      renderFromJson(contentItem, `${uniqKey}_${index}`, components),
    );
  }

  if (item instanceof Object) {
    if (typeof item.elemName !== "string") {
      return item;
    }

    const { elemName, children, ...otherProperties } = item;

    otherProperties.key ??= uniqKey;

    return React.createElement(
      getComponent(elemName, components),
      otherProperties,
      renderFromJson(children, "", components),
    );
  }

  return null;
}
```

Рисунок 28 – Код функции renderFromJson

На вход функция renderFromJson принимает 3 аргумента.

– item – это элемент схемы, предполагается передача простых объектов (набора ключей и значений), строк, чисел, null. Также можно передавать массив из вышеуказанных типов;
– uniqKey – уникальный ключ, использующийся для внутренних целей frontend-фреймворков для оптимизации обновления элементов, переданых в массиве. Автоматически генерируется при рекурсивном вызове в случае передачи массива;
– components – набор компонентов в виде простого js объекта, отображен на рисунке 29.

```
35  import { Link } from "../InteractiveControl/InteractiveControl";
36  import { Img } from "../Img/Img";
37
38  const components = {
39    Img: Img,
40    Link: Link,
41    // ...
42  };
```

Рисунок 29 – Пример объекта components

Рассмотрим содержание функции renderFromJson. Если в качестве item передается строка или число, просто возвращаем item. Если передан массив, производим рекурсивную обработку. Если же передан объект, проверяем свойство elemName (имя необходимого компонента) у этого объекта, если это не строка, возвращаем null, если же строка, то получаем компонент и возвращаем результат вызова React.createElement(...), который создаст компонент, а затем отрендерит фрагмент пользовательского интерфейса.

Выше мы рассмотрели все ключевые части демонстрационного веб-приложения с подробным описанием процесса их работы, начиная от запуска веб-приложения и до механизма рендеринга перед выводом в браузер.

3.3 Реализация механизма генерации структуры пользовательского интерфейса в формате JSON с использованием языка PHP

При необходимости хранения логики пользовательского интерфейса в серверной части веб-приложения с последующей ее передачей в клиентскую часть в формате JSON можно использовать 2 подхода. Можно просто хранить JSON файлы в том же виде, в котором они будут передаваться при необходимости на клиентскую часть. Но при таком подходе не будет удобной возможности их изменять в случае необходимости. Второй подход – это разработать удобный механизм генерации (далее по тексту – конструктор) структуры пользовательского интерфейса на языке реализации серверной части веб-приложения.

Далее будет показан пример реализации конструктора структуры пользовательского интерфейса в формате json на языке программирования PHP с использованием паттернов Фасад (англ. Facade) и Плавный строитель (англ. Fluent Builder).

«Шаблон фасад (англ. Facade) — структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.» [16]

«Паттерн Fluent Builder позволяет упростить процесс создания сложных объектов с помощью методов-цепочек, которые наделяют объект каким-то определенным качеством. Применение данного паттерна делает процесс конструирования объектов более прозрачным, а код более читабельным.» [23]

Учитывая, что на выходе мы должны получить JSON, логично первым делом создать базовый абстрактный класс BaseEntity для всех сущностей конструктора, который будет реализовывать механизм конвертации PHP объектов в json структуру. Код базового класса BaseEntity представлен на рисунке 30. В этот же базовый класс добавим свойство $_data, которое в

формате ассоциативного массива будет хранить все параметры. В дополнение к $_data добавим и методы по добавлению и получению параметров.

```php
abstract class BaseEntity implements JsonSerializable
{
  protected array $_data = [];

  protected function addParam(string $name, mixed $val, bool $when = true): static
  {
    if ($when) {
      $this->_data[$name] = $val;
    }

    return $this;
  }

  protected function addMultiParam(string $name, mixed $val, bool $when = true): static
  {
    if ($when) {
      if (empty($this->_data[$name]) || !is_array($this->_data[$name])) {
        $this->_data[$name] = [];
      }

      $this->_data[$name][] = $val;
    }

    return $this;
  }

  public function getParam(string $name): mixed
  {
    return $this->_data[$name] ?? null;
  }

  public function jsonSerialize(): ?array
  {
    return $this->_data;
  }
}
```

Рисунок 30 – Код класса BaseEntity

Далее создадим абстрактный базовый класс компонентов пользовательского интерфейса UiBase, код которого показан на рисунке 31. Ui в названии класса – это сокращение от user interface, что переводится на русский язык как пользовательский интерфейс.

UiBase содержит конструктор с приемом обязательного аргумента $elemName, на основе которого функция рендеринга в клиентской части определит какой именно компонент необходимо отрендерить. Также UiBase содержит метод для задания уникального идентификатора key, необходимого для для компонентов, передаваемых в массивах.

```php
abstract class UiBase extends BaseEntity
{
  /**
   * @param string $elemName
   */
  public function __construct(string $elemName)
  {
    $this->addParam('elemName', $elemName);
  }

  /**
   * Уникальный строковой идентификатор для компонентов, передаваемых в массивах
   * @param string $key
   * @return static
   */
  public function key(string $key): static
  {
    return $this->addParam('key', $key);
  }
}
```

Рисунок 31 – Код класса UiBase

От UiBase можно уже наследовать классы для компонентов, которые не могут содержать обычных дочерних компонентов. Например, компонент изображения, компонент значка, компонент перевода текста на новую строку. Ниже на рисунке 32 приведен код класса UiImage.

```php
class UiImage extends UiBase
{
  public function __construct()
  {
    parent::__construct('@Img');
  }

  public function src(string $url): static
  {
    return $this->addParam('src', $url);
  }

  public function alt(string $alt): static
  {
    return $this->addParam('alt', $alt);
  }

  public function title(string $title): static
  {
    return $this->addParam('title', $title);
  }
}
```

Рисунок 32 – Код класса UiImage

Для компонентов, которые могут содержать дочерние элементы, сделаем абстрактный класс UiBaseWithChildren, код которого показан на рисунке 33.

```php
abstract class UiBaseWithChildren extends UiBase
{
  public function children(UiBase|string|null ...$children): static
  {
    foreach ($children as $child) {
      if ($child === null || $child === '') {
        continue;
      }

      if (is_string($child)) {
        $child = html_entity_decode($child, ENT_HTML401 | ENT_QUOTES);
      }

      $this->addMultiParam('children', $child);
    }

    return $this;
  }
}
```

Рисунок 33 – Код класса UiBaseWithChildren

От UiBase и UiBaseWithChildren будут наследоваться классы для любых компонентов пользовательского интерфейса. В рамках разработки демонстрационного веб-приложения получилась иерархия классов, отображенная на рисунке 34.

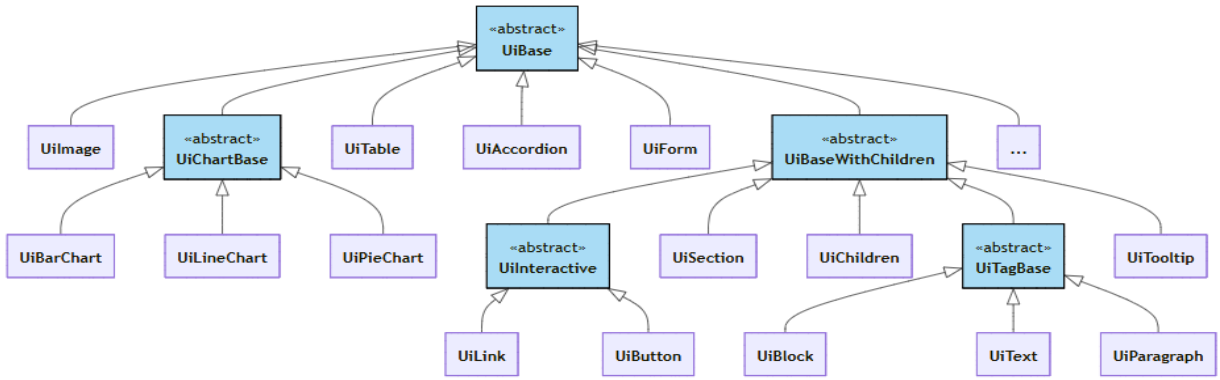

Рисунок 34 – Иерархия классов конструктора структуры пользовательского интерфейса

Каждый класс должен содержать методы для установки всех параметров, которые принимает компонент пользовательского интерфейса в клиентской части веб-приложения. Для примера рассмотрим компонент Icon (значок). На рисунке 35 представлен код компонента Icon в клиентской части.

```
import React from "react";
import { Home, Close, Help, Phone, Comment } from "@mui/icons-material";

const icons = { Home, Close, Help, Phone, Comment };

export default function Icon({ type, color, size }) {
  return icons[type]
    ? React.createElement(icons[type], { color, fontSize: size })
    : null;
}
```

Рисунок 35 – Код компонента Icon в клиентской части веб-приложения

Как видно на рисунке 35, компонент Icon принимает параметры type, color и size. Для параметра type предполагается прием следующих строковых значений Home, Close, Help, Phone и Comment, так как только значки с такими названиями импортируются из сторонней библиотеки и могут быть обработаны компонентом. Если передать любое другое значение, то компонент вернет null, а значит в месте отображения этого значка ничего не выведется.

Параметр size и color мы передаем напрямую в компонент сторонней библиотеки, а из документации к библиотеке мы узнаем, что в качестве size ожидается строковое значение inherit, large, medium, small. Для параметра color предусмотрены значения inherit, action, disabled, primary, secondary, error, info, success, warning.

Если аргумент принимает не любое строковое значение, логично использовать enum. Например, для типа значка будем использовать enum IconType, представленный на рисунке 36.

```php
enum IconType: string
{
  case HELP = 'Help';
  case HOME = 'Home';
  case CLOSE = 'Close';
  case COMMENT = 'Comment';
  case PHONE = 'Phone';
}
```

Рисунок 36 – Код enum IconType

Далее рассмотрим код класса UiIcon на рисунке 37.

```php
class UiIcon extends UiBase
{
  public function __construct()
  {
    parent::__construct('@Icon');
  }

  public function type(IconType $type): static
  {
    return $this->addParam('type', $type);
  }

  public function size(IconSize $size): static
  {
    return $this->addParam('size', $size);
  }

  public function color(IconColor $color): static
  {
    return $this->addParam('color', $color);
  }
}
```

Рисунок 37 – Код класса UiIcon

Для удобства использования компонентов создаем класс UI, который будет содержать набор статических методов для создания объектов каждого компонента. Код класса UI представлен на рисунке 38. Это как раз и будет фасад для всего конструктора пользовательского интерфейса.

```php
class UI
{
  public static function Section(): UiSection
  {
    return new UiSection();
  }

  public static function Link(?string $url = null): UiLink
  {
    return (new UiLink())->url($url);
  }

  public static function Icon(IconType $type): UiIcon
  {
    return (new UiIcon())->type($type);
  }
  // ...
```

Рисунок 38 – Код класса UI

При работе с конструктором через фасад UI в среде разработки PHPStorm [30] сразу можно увидеть все доступные компоненты в классе фасаде. Пример использования представлен на рисунке 39.

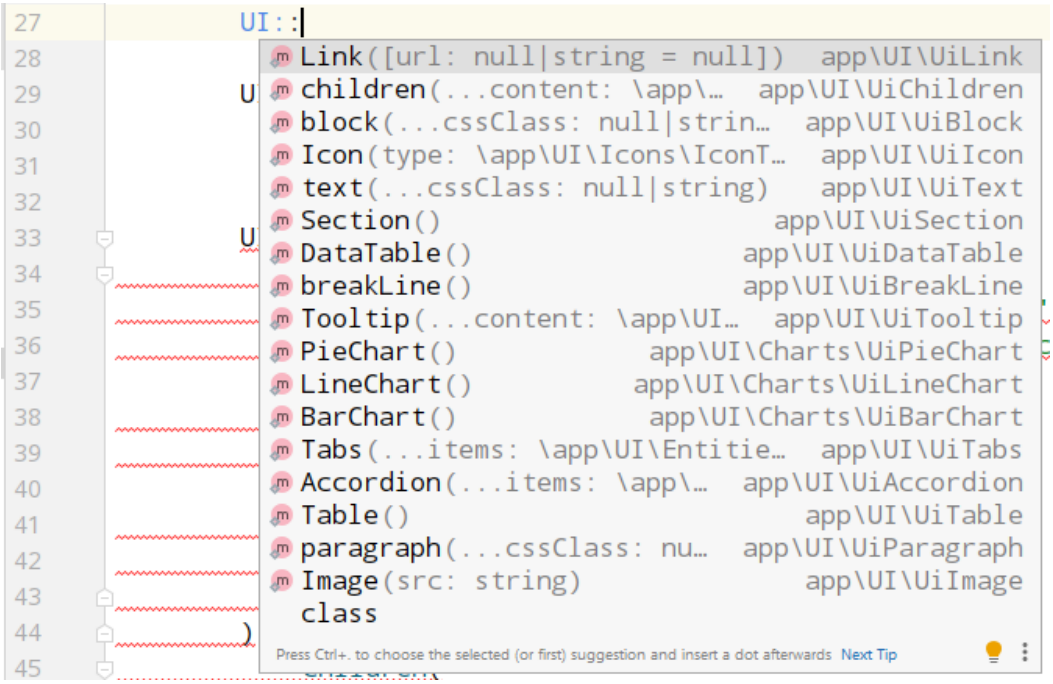

Рисунок 39 – Выпадающий список класса (фасада) UI в среде разработки

Теперь рассмотрим пример PHP кода со структурой пользовательского интерфейса на рисунке 40.

```php
public static function header(): UiBase
{
  return UI::Section()->children(
    UI::block(Css::ROW, Css::X_CENTER, Css::SPACE_15)->children(
      UI::Link('/')->children('Главная'),
      UI::Link('/profile')->children('Профиль'),
      UI::Link('/charts')->children('Графики'),
      UI::Link('/users')->children('Пользователи'),
      UI::Link('/404')->children('404'),

      UI::Link()
        ->action('contacts-modal')
        ->children('Окно контактов'),

      UI::Tooltip(
        UI::children(
          'Это демонстрационное веб-приложение, в котором' .
          'вся логика пользовательского интерфейса хранится в серверной части.',
          UI::breakLine(),
          'Можете посмотреть графики ',

          UI::Link('/charts')
            ->children('здесь'),
        )
      )
        ->children(
          UI::Icon(IconType::HELP)
            ->size(IconSize::MEDIUM)
            ->color(IconColor::PRIMARY)
        )
    ),
  );
}
```

Рисунок 40 – Код логики пользовательского интерфейса на языке PHP

На основе этого кода будут сгенерированы данные в формате JSON, отображенные на рисунке 41 и результат вывода в браузере на рисунке 42.

```json
{
  "elemName": "@Section",
  "children": [
    {
      "elemName": "div",
      "className": "flex-row flex-x-center gap15",
      "children": [
        { "elemName": "@Link", "url": "/", "children": ["Главная"] },
        { "elemName": "@Link", "url": "/profile", "children": ["Профиль"] },
        { "elemName": "@Link", "url": "/charts", "children": ["Графики"] },
        { "elemName": "@Link", "url": "/users", "children": ["Пользователи"] },
        { "elemName": "@Link", "url": "/404", "children": ["404"] },
        { "elemName": "@Link", "data-action": "contacts-modal", "children": ["Окно контактов"] },
        {
          "elemName": "@Tooltip",
          "tooltip": [
            {
              "elemName": "",
              "children": [
                "Это демонстрационное веб-приложение, в которомвся логика пользовательского ...",
                { "elemName": "br" },
                "Можете посмотреть графики ",
                { "elemName": "@Link", "url": "/charts", "children": ["здесь"] }
              ]
            }
          ],
          "children": [
            {
              "elemName": "@Icon",
              "type": "Help",
              "size": "medium",
              "color": "primary"
            }
          ]
        }
      ]
    }
  ]
}
```

Рисунок 41 – JSON данные стурктуры пользовательского интерфейса

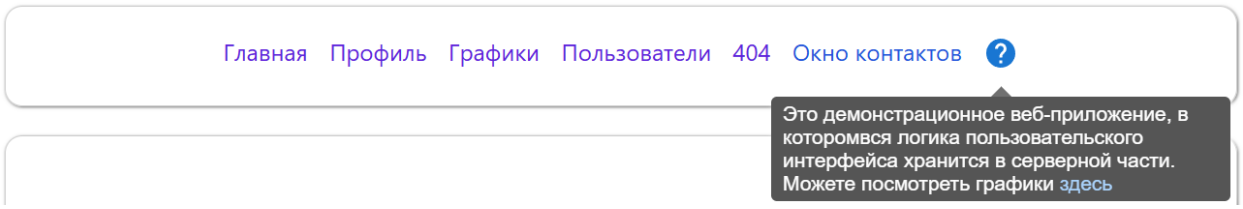

Рисунок 42 – Вывод в браузере фрагмента интерфейса, на основе кода генерации на рисунке 41

Аналогично конструктору для генерации общей структуры пользовательского интерфейса можно реализовать и генерацию логики для любых других более частных случаев. На рисунке 43 представлен фрагмент кода генерации логики веб-формы.

```
35      return new FormBaseItems(
36        Form::textField('name')
37          ->placeholder('Имя'),
38
39        Form::dateField('birthday')
40          ->placeholder('Дата рождения'),
41
42        Form::selectorField('activity')
43          ->placeholder('Ваше основное занятие')
44          ->options(
45            new FormSelectorOption(1, 'Студент'),
46            new FormSelectorOption(2, 'Наемный работник'),
47            new FormSelectorOption(3, 'Предприниматель'),
48            new FormSelectorOption(4, 'Безработный'),
49            new FormSelectorOption(5, 'Пенсионер'),
50          ),
51
52        Form::checkboxField('accept')
53          ->checkboxLabel(
54            'Я соглашаюсь на обработку персональной информации и с ',
55
56            UI::Link('/privacy-policy')
57              ->cssClass(Css::INLINE_BLOCK)
58              ->children('Политикой конфиденциальности'),
59          )
60          ->value(true),
61
62        Form::submit('Сохранить'),
63      );
```

Рисунок 43 – Фрагмент кода генерации структуры полей веб-формы

В данном случае фасадом является класс Form, все элементы формы имеют общего родителя FormBaseItem, который является потомком BaseEntity.

Использование вышеописанного конструктора вместо просто хранения JSON дает нам ряд преимуществ, главное из которых – это использование

любых возможностей языка программирования (ооп, условия, циклы, константы, перечисления, проверку типов и так далее). Вторым немаловажным преимуществом является наличие в средах разработки выпадающих подсказок со списком вариантов автоматического завершения кода, в котором содержатся все доступные методы указанного объекта или класса (если это Фасад), что позволяет быстро выбрать подходящий метод даже если разработкой занимается специалист, поверхностно знакомый с проектом.

Выводы по главе 3

Логика пользовательского интерфейса веб-приложения может быть представлены в виде древовидной структуры данных и может передаваться между серверной и клиентской частями веб-приложения в формате JSON;

В клиентской части веб-приложения можно полностью исключить размещение логики пользовательского интерфейса, получая данные о ее структуре с сервера для каждой страницы и затем формируя на ее основе разметку веб-страницы;

Для генерации структуры пользовательского интерфейса со всей ее внутренней логикой можно разработать удобный конструктор в серверной части веб-приложения с применением паттернов проектирования Фасад (англ. Facade) и Плавный строитель (англ. Fluent Builder).

Глава 4 Оценка эффективности модели веб-приложения с размещением логики пользовательского интерфейса в серверной части

4.1 Определение рекомендательных показателей объема файлов веб-приложений

Как уже было написано выше, в основном проблемы с медленной работой объемных веб-приложений возникают на мобильных устройствах с малым объемом оперативной памяти и(или) при использовании мобильного интернет-соединении с низкой скоростью.

Если посмотреть статистику данных по скорости мобильного интернета в Российской Федерации [32], то на декабрь 2024 года можно было увидеть в среднем 26 Мбит/сек. Тут нужно отметить, что средняя скорость разная в зависимости от региона. Так удалось найти данные на конец 2023 года, в которых указано, что в Москве средняя скорость была 63 Мбита в секунду, а в других регионах всего лишь 18,7 [7]. Причем, нужно понимать, что 18,7 – это средняя скорость вне Москвы, у одного пользователя может быть скорость интернет-соединения 30 Мбит/сек, а у другого – 6, в среднем получится – 18. В интернете можно найти множество негативных комментариев/отзывов людей о мобильном интернете от разных операторов, поэтому мы понимаем, что в отдельных довольно разнообразных случаях все же есть проблемы.

Наглядно увидеть качество мобильного интернет-соединения можно на карте веб-сервиса nperf.com [27], там можно выбрать любого оператора, но результаты примерно одинаковые по всем операторам. Скриншот карты приведен на рисунке 44.

Как видим, синих точек не так уж и мало, это точки, в которых при замере скорости интернет-соединения у абонента мобильного оператора составила от 0 до 10 мбит/сек. На средний показатель 5 Мбит/сек мы и будем ориентироваться при расчетах максимального объема кода клиентской части

веб-приложения во время загрузки. При скорости интернета 5 мбит/сек за секунду загружается около 0,6 мбайт данных.

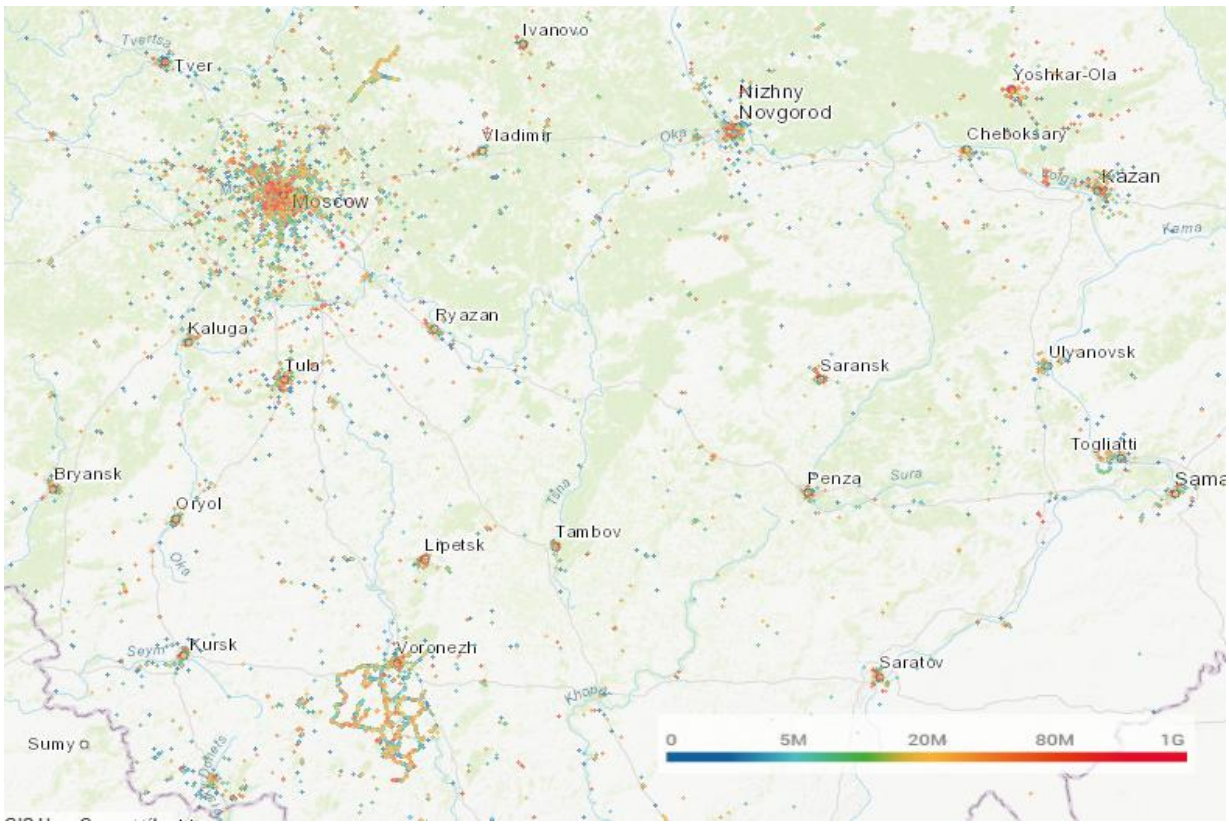

Рисунок 44 – Карта с отметками скорости мобильного интернета

При поиске информации по рекомендуемому времени загрузки веб-сайтов множество ненаучных источников указывают показатель в 2-3 секунды, но не удалось найти ссылку на какие-либо исследования по этому вопросу. Инструмент LightHouse, встроенный в сервисе Google PageSpeed, использующийся для оценки скорости загрузки сайтов по всему миру, при оценке скорости загрузки веб-сайтов считает сайт быстро загружающимся, если время загрузки находится в пределах от 0 до 3,8 секунды [33]. На этот показатель в 3,8 сек мы и будем ориентироваться. Стоит также учитывать наличие сетевых задержек, времени поиска DNS, подключение к серверу, установку защищенного соединения. Для всех этих расходов определим фиксированное значение в 0,4 сек, основываясь на реальных данных при

запросе страницы демонстрационного веб-приложения, представленных на рисунке 45.

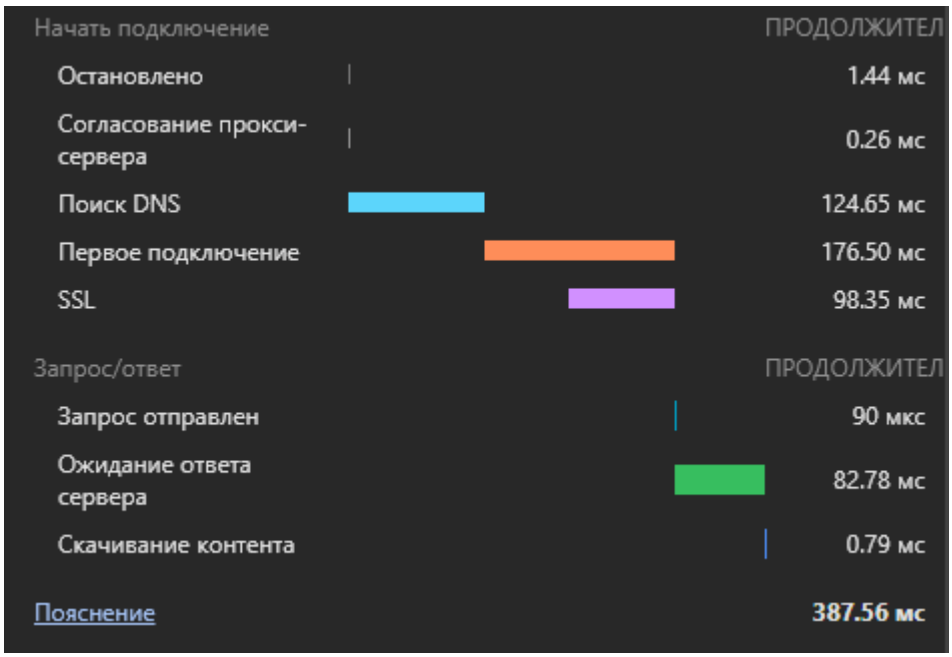

Рисунок 45 – Расход времени на выполнение запроса к серверу

Что касается использования оперативной памяти веб-приложением, то выделить какие-то рамки здесь довольно трудно, так как доступный объем для вкладки с работающим веб-приложением будет зависеть не только от размера памяти на самом устройстве, но также и от количества открытых в текущий момент приложений, а также от количества открытых вкладок в браузере пользователя. В целом мы не ошибемся, если скажем, что чем меньше памяти используется, тем меньше риск упереться в доступный ее объем. При нехватке оперативной памяти вкладка браузера с веб-приложением может просто прекратить работу, выдав соответствую ошибку. В декабре 2024 года время еще присутствуют в продаже недорогие смартфоны с 2 гигабайтами оперативной памяти, и в использовании такие слабые телефонов тоже есть, ведь не все могут себе позволить менять телефон раз в пару лет. В целом наличие проблемы подтверждает немалое количество статей и видео по теме торможения браузера на мобильном.

Для определения влияния размера веб-приложения на расходуемый объем оперативной памяти вкладкой браузера было проанализировано 39 веб-сайтов/веб-приложений, данные замеров были занесены в электронную таблицу и построена диаграмма, представленная на рисунке 46.

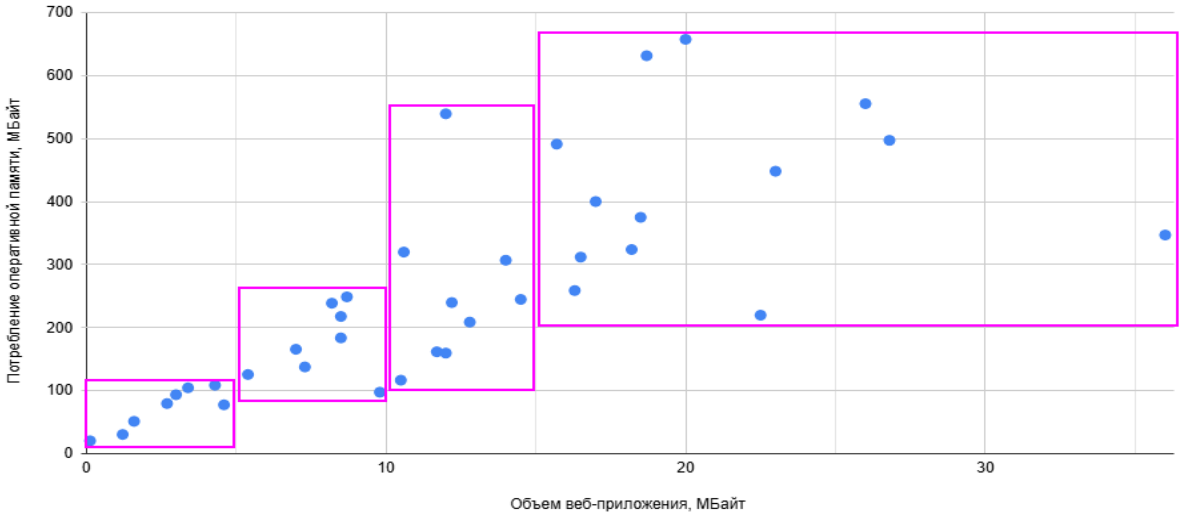

Рисунок 46 – Потребление оперативной памяти относительно объема веб-страницы

После группировки полученных данных была заполнена таблица 3. Как видим, существует не четкая, но прямая зависимость между объемом веб-страницы и количеством оперативной памяти, потребляемым вкладкой браузера.

Таблица 3 – Группировка данных по объему веб-страницы

Показатель	Значение показателя			
Объем веб-страницы, Мбайт	0-5	5-10	10-15	15+
Используемая оперативная память, Мбайт	0-105	98-250	110-540	260+

При оценке будем иметь в виду, чтобы общий размер файлов веб-приложения не превышал 5 Мбайт. Примерно такой объем файлов с учетом gzip сжатия (уменьшает расходуемый трафик в 2-3 раза) сможет загрузиться

за 3,8 секунды при скорости интернет-соединения 5 Мбит/сек с учетом задержки на установку соединения с сервером 0,4 сек. Веб-приложение размером 5 Мбайт и оперативной памяти будет расходовать около 100 Мбайт, поэтому будет нормально работать даже на бюджетных или старых устройствах с минимальным объемом оперативной памяти.

4.2 Оценка изменения времени загрузки и расхода оперативной памяти веб-приложения при использовании предложенной модели

Корпоративные веб-приложения могут содержать разное количество логик пользовательского интерфейса. В малофункциональных веб-приложениях общий код компонентов может занимать 90%, а 10% будет составлять бизнес логика, в многофункциональном веб-приложении ситуация может быть прямо противоположная и бизнес-логика пользовательского интерфейса будет занимать 90%. Поэтому и возможности для ее сокращения будут совершенно различными для разных проектов. Соответственно влияние на скорость загрузки и расход оперативной памяти тоже будет разным.

Чтобы понимать, сколько логики пользовательского интерфейса может быть в веб-приложении в целом и на одной странице в частности, можно посмотреть проекты с открытым исходным кодом. В качестве примера рассмотрим код клиентской части административной панели системы управления интернет-магазином OpenCart [29]. Общий размер файлов всех шаблонов страниц административной панели OpenCart составляет 1,34 Мбайта, это довольно небольшое веб-приложение с самым необходимым функционалом. Всего административная панель содержит около 100 страниц, подавляющее большинство которых очень простые, например таблица на 3-5 колонок или веб-форма, включающая 5-10 полей. Но есть и несколько страниц, содержащих сложную логику отображения и взаимодействия, и потому имеющими относительно большой размер файлов, включающих в себя разметку и код скрипта:

– страница информации о заказе – 153 Кбайт;
– страница с формой данных о товаре – 146 Кбайт;
– страница с формой общих настроек интернет-магазина – 99 Кбайт;
– страница с формой данными о клиенте – 74 Кбайт;
– страница с формой данных о партнере – 31 Кбайт.

Таким образом можем определить, что средний размер логики пользовательского интерфейса относительно сложных страниц около 100 Кбайт. В корпоративных веб-приложениях могут быть десятки и даже сотни страниц, наполненных сложной логикой отображения и взаимодействия.

Разработать настоящее корпоративное веб-приложения в рамках научно-исследовательской работы не представляется возможным ввиду огромной трудоемкости такого дела, но мы можем в наше демонстрационное веб-приложение добавлять произвольный javascript код для увеличения размера, после чего произвести замеры скорости загрузки и расхода оперативной памяти. Произведем замеры для трех вариантов веб-приложений разных размеров, представленных в таблице 4.

Таблица 4 - Экспериментальные веб-приложения разных размеров

Условный размер веб-приложения	Малое	Среднее	Большое	Демонстрационное
Эквивалент количества веб-страниц со сложной логикой	30	90	180	не ограничено
Объем логики, Мбайт	3	9	18	0,1
Объем логики с учетом минификации, Мбайт	1	3	6	0,03

Замеры показателей производились на ноутбуке с установленной операционной системой Windows 11 в веб-браузере Google Chrome 118 с ограничением скорости интернет-соединения 5 Мбит/сек. Результаты замеров представлены в таблице 5.

Таблица 5 – Сравнение показателей 3-х веб-приложений разных размеров

Показатель	Малое веб-приложение	Среднее веб-приложение	Большое веб-приложение
Размер кода общих компонентов, МБайт	1,23	1,23	1,23
Размер логики интерфейса в минифицированном javascript файле, МБайт	1	3	6
Общий размер файлов клиентской части веб-приложения, МБайт	2,23	4,23	7,23
Объем интернет-трафика с gzip-сжатием, МБайт	1,0	1,6	2,5
Время загрузки, сек	2,1	3,2	4,6

Мы получили вполне ожидаемые результаты. Чем больше размер веб-приложения, тем больше времени оно требует для загрузки и больше оперативной памяти используем в процессе своей работы. Теперь же замеряем показатели демонстрационного веб-приложения с добавлением 100 Кбайт JSON данных структуры пользовательского интерфейса условно сложной страницы на начальный html файл. Результаты добавим в таблицу 6. Также добавим разницу в процентах для каждого варианта веб-приложения по сравнению с демонстрационным веб-приложением.

Таблица 6 – Сравнение показателей с демонстрационным веб-приложением

Показатель	Малое веб-приложение	Среднее веб-приложение	Большое веб-приложение	Демонстрационное веб-приложение
Размер кода общих компонентов, МБайт	1,23	1,23	1,23	1,23
Размер логики интерфейса в минифицированном javascript файле, МБайт	1	3	6	0,1 (включено в html файл)
Общий размер файлов клиентской части веб-приложения, МБайт	2,23 (-40%)	4,23 (-69%)	7,23 (-82%)	1,33
Объем интернет-трафика с gzip-сжатием, МБайт	1,0 (-42%)	1,6 (-64%)	2,5 (-77%)	0,58
Время загрузки, сек	2,1 (-28%)	3,2 (-53%)	4,6 (-67%)	1,5
Потребление оперативной памяти, МБайт	36 (-11%)	45 (-29%)	54 (-41%)	32

Как видно из результатов в таблице 6, чем больше логики пользовательского интерфейса содержится в клиентской части веб-приложения, тем больший эффект дает применение предложенной модели.

Для веб-приложений, содержащих от 3 до 18 Мбайт (в минифицированном виде от 1 до 6 Мбайт) логики пользовательского интерфейса при размере общих компонентов 1,23 Мбайта время загрузки при скорости интернет соединения 5 Мбит/сек сокращается на 28-67%, а объем потребляемой оперативной памяти уменьшается на 11-41%. Это является хорошим результатом и подтверждением сформулированный гипотезы исследования.

Также следует отметить, что демонстрационное веб-приложение соответствует рекомендуемым показателям размера и времени загрузки, определенным в параграфе 4.1 и имеет запас для их увеличения.

В большинстве случаев корпоративные веб-приложения состоят из множества страниц, содержащих списки разных элементов, формы для редактирования отдельных элементов и отчеты. Основными компонентами корпоративных веб-приложений являются таблицы, формы и графики. Применение модели размещения логики пользовательского интерфейса в серверной части веб-приложения для конкретного проекта, в том числе корпоративного, будет актуальна в случае, если клиентская часть веб-приложения реализована по принципу одностраничного приложения с применением современных javascript фреймворков/библиотек, таких как React и Vue и если клиентская часть уже имеет (или будет иметь) немалый размер и если в клиентской части уже есть (или будет) много разной логики отображения и взаимодействия с пользователем, которая поддается унификации.

Выводы по главе 4

Опираясь на данные о скорости мобильного интернета в Российской Федерации за пределами Москвы, в качестве эталонной скорости интернет-соединения для замера времени загрузки веб-приложения был определен уровень 5 Мбит/сек.

Основываясь на материалах в сети интернет в целом и на градации сайтов по времени загрузки в инструменте Lighthouse [33], в качестве эталонного времени загрузки веб-приложения был определен уровень в 3,8 секунды.

На основании эталонных показателей скорости интернета и времени загрузки был определен максимальный размер интернет-трафика в 2 Мбайта с учетом gzip-сжатия и сетевой задержки 0,3 сек и максимальный стартовый размер веб-приложения – 5 Мбайт. В такой объем должны входить файлы html, css и javascript. Демонстрационное веб-приложение имеет стартовый размер 1,33 Мбайта и при скорости интернет-соединения 5 Мбит/сек грузится за 1,5 секунд, что на 28-67% меньше тестовых веб-приложений, и потребляет 32 Мбайта оперативной памяти, что на 11-41% меньше, чем в тестовых веб-приложениях разного размера. Полученные результаты свидетельствуют и возможности комфортной работы с веб-приложением на бюджетных или старых устройствах с минимальным объемом оперативной памяти и низкой скоростью доступа в интернет.

Заключение

В ходе данной научно-исследовательской работы рассматривались архитектурные особенности при построении легковесных веб-приложений, включая корпоративные. В первой главе было представлен теоретический обзор отрасли веб-приложений и краткой истории ее развития. Вторая глава посвящена исследованию изученности вопроса архитектуры веб-приложений. В процессе проведения анализа научной литературы было выявлено более 30 источников, которые так или иначе затрагивают общие архитектурные аспекты и способы оптимизации, в том числе и сокращения размера веб-приложений. Но последнему вопросу уделяется довольно мало внимания.

Проанализировав имеющиеся в научных публикациях способы сокращения размера веб-приложений, был выработан альтернативный способ сокращения размера клиентской части веб-приложений, работающих по принципу одностраничного приложения. Суть предложенной модели заключается в переносе логики пользовательского интерфейса из клиентской части в серверную и ее передаче в формате JSON на клиентскую часть при необходимости, например, при переходе на другую страницу, открытии модального окна и так далее.

В третьей главе было дано подробное теоретическое описание предложенной модели, после чего было детально описано разработанное демонстрационное веб-приложение, реализующее ее использование. Далее была показана реализация конструктора (механизма генерации) структуры пользовательского интерфейса в формате JSON с использованием языка PHP в серверной части веб-приложения.

В четвертой главе были определены показатели времени загрузки и объема файлов веб-приложений, которые желательны к соблюдению для комфортной работы веб-приложения при низкой скорости интернет-соединения и/или на устройствах с низким объемом оперативной памяти. После определения показателей была произведена оценка их соответствию

демонстрационного веб-приложения, а также степень влияния на время загрузки и потребление оперативной памяти веб-приложениями разных размеров при использовании предложенной модели. Гипотеза исследования была подтверждена.

Обобщая результаты проделанной работы, сформулируем краткие выводы: общеизвестные методы оптимизации размера веб-приложений, описанные в научных публикациях, и модель размещения логики пользовательского интерфейса в серверной части веб-приложения, выработанный в ходе данной научной работы, позволяют уменьшать размер клиентской части веб-приложения, что в свою очередь приводит к ускорению его загрузки и снижению расхода оперативной памяти на клиентских устройствах. Демонстрационное веб-приложение, разработанное с использованием модели размещения логики пользовательского интерфейса в серверной части веб-приложения, имеет стартовый размер 1,33 Мбайта, загружается за 1,5 секунды при скорости интернет-соединения 5 Мбит/сек и потребляет 32 Мбайта оперативной памяти, что позволяет комфортно работать с веб-приложением на устройствах с малым объемом оперативной памяти и(или) при низкой скоростью доступа в интернет.

Научная значимость проведенного исследования заключается в выработке модели, представляющей альтернативный архитектурный подход к созданию веб-приложений. Практическая значимость модели заключается в возможности снижения нагрузки и требований к ресурсам клиентских устройств. Научная значимость примеров реализации заключается в возможностях улучшения процесса разработки за счет увеличения декларативности кода, описывающего логику пользовательского интерфейса. Практическая значимость примеров реализации на серверной и клиентской стороне заключается в повышении читаемости кода и уменьшения возможности появления ошибок за счет строгих ограничений и типизации сущностей, которыми может оперировать разработчик в процессе описания структуры пользовательского интерфейса.

Список используемой литературы и используемых источников

1. Авчинников, В. В. Проблемы размещения бизнес-логики в клиентской части веб-приложений / В. В. Авчинников. — Текст : непосредственный // Молодой ученый. — 2024. — № 18 (517). — С. 11-13. — URL: https://moluch.ru/archive/517/111927/ (дата обращения: 23.11.2024).

2. Авчинников, В. В. Сокращение бизнес-логики в клиентской части веб-приложений / В. В. Авчинников. — Текст : непосредственный // Исследования молодых ученых : материалы XC Междунар. науч. конф. (г. Казань, ноябрь 2024 г.). — Казань : Молодой ученый, 2024. — С. 1-5. — URL: https://moluch.ru/conf/stud/archive/524/18706/ (дата обращения: 23.11.2024).

3. Вершинин, Е. В. Сравнение различных архитектурных решений при разработке веб-приложения / Е. В. Вершинин, А. Е. Хромов // Наука. Исследования. Практика : сборник избранных статей по материалам Международной научной конференции, Санкт-Петербург, 25 апреля 2022 года. – Санкт-Петербург: Частное научно-образовательное учреждение дополнительного профессионального образования Гуманитарный национальный исследовательский институт «НАЦРАЗВИТИЕ», 2022. – С. 46-49. – EDN BOCURR.

4. Горбачев, А. А., Горбачева, Е. С. Сравнение классического процесса реализации веб-приложений и подхода с использованием библиотеки React // Молодой исследователь Дона. 2020. №1 (22). URL: https://cyberleninka.ru/article/n/sravnenie-klassicheskogo-protsessa-realizatsii-veb-prilozheniy-i-podhoda-s-ispolzovaniem-biblioteki-react (дата обращения: 19.12.2024).

5. Гридин, В. Н. Методы повышения производительности современных веб-приложений / В. Н. Гридин, В. И. Анисимов, С. А. Васильев // Известия ЮФУ. Технические науки. – 2020. – № 2(212). – С. 193-200. – DOI 10.18522/2311-3103-2020-2-193-200. – EDN BGUPQL.

6. Исследование сжатия WebP / [Электронный ресурс] // Google for Developers : [сайт]. — URL: https://developers.google.com/speed/webp/docs/webp_study?hl=ru (дата обращения: 27.01.2025).

7. Кодачигов, В. Скоростной интернет, самая высокая скорость интернета в Москве / Кодачигов В. [Электронный ресурс] // Известия : [сайт]. — URL: https://iz.ru/1602607/valerii-kodachigov/razgruzka-sdannykh-v-regionakh-snizhaetsia-skorost-mobilnogo-interneta (дата обращения: 27.01.2025).

8. Логинова, Н. В. Методы оптимизации производительности web-приложений // Наука и образование сегодня. 2024. №1 (78). URL: https://cyberleninka.ru/article/n/metody-optimizatsii-proizvoditelnosti-web-prilozheniy (дата обращения: 19.12.2024).

9. Медведев, Ю. С. Увеличение скорости загрузки web-приложений / Ю. С. Медведев, В. В. Терехов // Успехи современной науки. – 2017. – Т. 6, № 3. – С. 181-185. – EDN YNFORV.

10. Оптимизация веб-страницы: подробное руководство / [Электронный ресурс] // Библиотека программиста : [сайт]. — URL: https://proglib.io/p/web-optimization (дата обращения: 19.12.2024).

11. Сайт / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/%D0%A1%D0%B0%D0%B9%D1%82 (дата обращения: 09.12.2024).

12. Сайт Google Fonts / [Электронный ресурс] // Google Fonts : [сайт]. — URL: https://fonts.google.com/ (дата обращения: 23.01.2025).

13. Сайт Terser / [Электронный ресурс] // Terser : [сайт]. — URL: https://terser.org/ (дата обращения: 23.01.2025).

14. Сайт Transfonter / [Электронный ресурс] // Transfonter : [сайт]. — URL: https://transfonter.org/ (дата обращения: 27.01.2025).

15. Статников, А. С. Особенности клиент-серверной архитектуры при реализации одностраничного web-приложения / А. С. Статников, Д. А. Фролов // Решетневские чтения : Материалы XXV Международной научно-

практической конференции, посвященной памяти генерального конструктора ракетно-космических систем академика М.Ф. Решетнева. В 2-х частях, Красноярск, 10–12 ноября 2021 года / Под общей редакцией Ю.Ю. Логинова. Том Часть 2. – Красноярск: Федеральное государственное бюджетное образовательное учреждение высшего образования "Сибирский государственный университет науки и технологий имени академика М.Ф. Решетнева", 2021. – С. 406-407. – EDN JMEFDL.

16. Фасад (шаблон проектирования) / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D1%81%D0%B0%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) (дата обращения: 27.01.2025).

17. Чернецкий, И. И. Анализ фреймворков для разработки современных веб-приложений / И. И. Чернецкий // Приоритетные направления развития науки в современном мире : Сборник научных статей по материалам XII Международной научно-практической конференции, Уфа, 22 августа 2023 года. – Уфа: Общество с ограниченной ответственностью "Научно-издательский центр "Вестник науки", 2023. – С. 244-253. – EDN GCRUGQ.

18. Яровая, Е. В. Нестандартные архитектура в написание веб приложений // Столыпинский вестник. 2022. №5. URL: https://cyberleninka.ru/article/n/nestandartnye-arhitektura-v-napisanie-veb-prilozheniy (дата обращения: 19.12.2024).

19. Clean-css: fast and efficient CSS optimizer / [Электронный ресурс] // Clean-css : [сайт]. — URL: https://clean-css.github.io/ (дата обращения: 23.01.2025).

20. Closure Compiler / [Электронный ресурс] // Google for Developers : [сайт]. — URL: https://developers.google.com/closure/compiler?hl=ru (дата обращения: 23.01.2025).

21. Css / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/CSS (дата обращения: 09.12.2024).

22. CssNano / [Электронный ресурс] // CssNano : [сайт]. — URL: https://cssnano.github.io/cssnano/ (дата обращения: 23.01.2025).

23. Fluent Builder / [Электронный ресурс] // METANIT.COM - Сайт о программировании : [сайт]. — URL: https://metanit.com/sharp/patterns/6.1.php (дата обращения: 27.01.2025).

24. JSON / [Электронный ресурс] // Википедия : [сайт]. — URL: https://ru.wikipedia.org/wiki/JSON (дата обращения: 27.01.2025).

25. JavaScript minification benchmarks: esbuild, terser, uglifyjs and other tools / [Электронный ресурс] // Minify JS Online : [сайт]. — URL: https://minify-js.com/benchmarks/ (дата обращения: 23.01.2025).

26. Lazy loading - Web Performance / [Электронный ресурс] // MDN : [сайт]. — URL: https://developer.mozilla.org/ru/docs/Web/Performance/Lazy_loading (дата обращения: 27.01.2025).

27. MTS Mobile 3G / 4G / 5G битрейт / [Электронный ресурс] // nPerf.com : [сайт]. — URL: https://www.nperf.com/ru/map/RU/-/157235.MTS-Mobile/download?ll=50.09152801558897&lg=41.01528051509887&zoom=3 (дата обращения: 27.01.2025).

28. Material UI: React components that implement Material Design / [Электронный ресурс] // MUI: The React component library you always wanted : [сайт]. — URL: https://mui.com/material-ui/ (дата обращения: 27.01.2025).

29. OpenCart - Open Source Shopping Cart Solution / [Электронный ресурс] // OpenCart : [сайт]. — URL: https://www.opencart.com (дата обращения: 27.01.2025).

30. PhpStorm: The PHP IDE by JetBrains / [Электронный ресурс] // JetBrains: Essential tools for software developers and teams : [сайт]. — URL: https://www.jetbrains.com/phpstorm/ (дата обращения: 27.01.2025).

31. React - the library for web and native user interfaces / [Электронный ресурс] // React : [сайт]. — URL: https://react.dev/ (дата обращения: 27.01.2025).

32. Russia's Mobile and Broadband Internet Speeds / [Электронный ресурс] // Speedtest : [сайт]. — URL: https://www.speedtest.net/global-index/russia (дата обращения: 27.01.2025).

33. Time to Interactive | Lighthouse / [Электронный ресурс] // Chrome for Developers : [сайт]. — URL: https://developer.chrome.com/docs/lighthouse/performance/interactive (дата обращения: 27.01.2025).

34. UglifyJS — JavaScript parser, compressor, minifier written in JS / [Электронный ресурс] // Lisperator.net — to create, to ilisperate : [сайт]. — URL: https://lisperator.net/uglifyjs/ (дата обращения: 23.01.2025).

35. Usage Statistics and Market Share of Server-side Programming Languages for Websites, December 2024 / [Электронный ресурс] // W3techs : [сайт]. — URL: https://w3techs.com/technologies/overview/programming_language (дата обращения: 09.12.2024).